INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

BLUE BOOK

VOLUME X – FASCICLE X.4

ANNEX F.2 TO RECOMMENDATION Z.100: SDL FORMAL DEFINITION

STATIC SEMANTICS

IXTH PLENARY ASSEMBLY
MELBOURNE, 14-25 NOVEMBER 1988

Geneva 1989

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

BLUE BOOK

VOLUME X – FASCICLE X.4

ANNEX F.2 TO RECOMMENDATION Z.100: SDL FORMAL DEFINITION

STATIC SEMANTICS

IXTH PLENARY ASSEMBLY
MELBOURNE, 14-25 NOVEMBER 1988

Geneva 1989

ISBN 92-61-03781-X

Printed in Switzerland

CONTENTS OF THE CCITT BOOK
APPLICABLE AFTER THE NINTH PLENARY ASSEMBLY (1988)

BLUE BOOK

FASCICLE III.6 – Line transmission of non-telephone signals. Transmission of sound-programme and television signals. Series H and J Recommendations (Study Group XV).

FASCICLE III.7 – Integrated Services Digital Network (ISDN) – General structure and service capabilities. Recommendations I.110-I.257 (Study Group XVIII).

FASCICLE III.8 – Integrated Services Digital Network (ISDN) – Overall network aspects and functions, ISDN user-network interfaces. Recommendations I.310-I.470 (Study Group XVIII).

FASCICLE III.9 – Integrated Services Digital Network (ISDN) – Internetwork interfaces and maintenance principles. Recommendations I.500-I.605 (Study Group XVIII).

Volume IV

FASCICLE IV.1 – General maintenance principles: maintenance of international transmission systems and telephone circuits. Recommendations M.10-M.782 (Study Group IV).

FASCICLE IV.2 – Maintenance of international telegraph, phototelegraph and leased circuits. Maintenance of the international public telephone network. Maintenance of maritime satellite and data transmission systems. Recommendations M.800-M.1375 (Study Group IV).

FASCICLE IV.3 – Maintenance of international sound-programme and television transmission circuits. Series N Recommendations (Study Group IV).

FASCICLE IV.4 – Specifications for measuring equipment. Series O Recommendations (Study Group IV).

Volume V – Telephone transmission quality. Series P Recommendations (Study Group XII).

Volume VI

FASCICLE VI.1 – General Recommendations on telephone switching and signalling. Functions and information flows for services in the ISDN. Supplements. Recommendations Q.1-Q.118 *bis* (Study Group XI).

FASCICLE VI.2 – Specifications of Signalling Systems Nos. 4 and 5. Recommendations Q.120-Q.180 (Study Group XI).

FASCICLE VI.3 – Specifications of Signalling System No. 6. Recommendations Q.251-Q.300 (Study Group XI).

FASCICLE VI.4 – Specifications of Signalling Systems R1 and R2. Recommendations Q.310-Q.490 (Study Group XI).

FASCICLE VI.5 – Digital local, transit, combined and international exchanges in integrated digital networks and mixed analogue-digital networks. Supplements. Recommendations Q.500-Q.554 (Study Group XI).

FASCICLE VI.6 – Interworking of signalling systems. Recommendations Q.601-Q.699 (Study Group XI).

FASCICLE VI.7 – Specifications of Signalling System No. 7. Recommendations Q.700-Q.716 (Study Group XI).

FASCICLE VI.8 – Specifications of Signalling System No. 7. Recommendations Q.721-Q.766 (Study Group XI).

FASCICLE VI.9 – Specifications of Signalling System No. 7. Recommendations Q.771-Q.795 (Study Group XI).

FASCICLE VI.10 – Digital subscriber signalling system No. 1 (DSS 1), data link layer. Recommendations Q.920-Q.921 (Study Group XI).

FASCICLE VI.11 – Digital subscriber signalling system No. 1 (DSS 1), network layer, user-network management. Recommendations Q.930-Q.940 (Study Group XI).

FASCICLE VI.12 – Public land mobile network. Interworking with ISDN and PSTN. Recommendations Q.1000-Q.1032 (Study Group XI).

FASCICLE VI.13 – Public land mobile network. Mobile application part and interfaces. Recommendations Q.1051-Q.1063 (Study Group XI).

FASCICLE VI.14 – Interworking with satellite mobile systems. Recommendations Q.1100-Q.1152 (Study Group XI).

Volume VII

FASCICLE VII.1 – Telegraph transmission. Series R Recommendations. Telegraph services terminal equipment. Series S Recommendations (Study Group IX).

FASCICLE VII.2 – Telegraph switching. Series U Recommendations (Study Group IX).

FASCICLE VII.3 – Terminal equipment and protocols for telematic services. Recommendations T.0-T.63 (Study Group VIII).

FASCICLE VII.4 – Conformance testing procedures for the Teletex Recommendations. Recommendation T.64 (Study Group VIII).

FASCICLE VII.5 – Terminal equipment and protocols for telematic services. Recommendations T.65-T.101, T.150-T.390 (Study Group VIII).

FASCICLE VII.6 – Terminal equipment and protocols for telematic services. Recommendations T.400-T.418 (Study Group VIII).

FASCICLE VII.7 – Terminal equipment and protocols for telematic services. Recommendations T.431-T.564 (Study Group VIII).

Volume VIII

FASCICLE VIII.1 – Data communication over the telephone network. Series V Recommendations (Study Group XVII).

FASCICLE VIII.2 – Data communication networks: services and facilities, interfaces. Recommendations X.1-X.32 (Study Group VII).

FASCICLE VIII.3 – Data communication networks: transmission, signalling and switching, network aspects, maintenance and administrative arrangements. Recommendations X.40-X.181 (Study Group VII).

FASCICLE VIII.4 – Data communication networks: Open Systems Interconnection (OSI) – Model and notation, service definition. Recommendations X.200-X.219 (Study Group VII).

FASCICLE VIII.5 – Data communication networks: Open Systems Interconnection (OSI) – Protocol specifications, conformance testing. Recommendations X.220-X.290 (Study Group VII).

FASCICLE VIII.6 – Data communication networks: interworking between networks, mobile data transmission systems, internetwork management. Recommendations X.300-X.370 (Study Group VII).

FASCICLE VIII.7 – Data communication networks: message handling systems. Recommendations X.400-X.420 (Study Group VII).

FASCICLE VIII.8 – Data communication networks: directory. Recommendations X.500-X.521 (Study Group VII).

Volume IX – Protection against interference. Series K Recommendations (Study Group V). Construction, installation and protection of cable and other elements of outside plant. Series L Recommendations (Study Group VI).

Volume X

FASCICLE X.1 – Functional Specification and Description Language (SDL). Criteria for using Formal Description Techniques (FDTs). Recommendation Z.100 and Annexes A, B, C and E, Recommendation Z.110 (Study Group X).

FASCICLE X.2 – Annex D to Recommendation Z.100: SDL user guidelines (Study Group X).

FASCICLE X.3 – Annex F.1 to Recommendation Z.100: SDL formal definition. Introduction (Study Group X).

FASCICLE X.4 – Annex F.2 to Recommendation Z.100: SDL formal definition. Static semantics (Study Group X).

FASCICLE X.5 – Annex F.3 to Recommendation Z.100: SDL formal definition. Dynamic semantics (Study Group X).

FASCICLE X.6 – CCITT High Level Language (CHILL). Recommendation Z.200 (Study Group X).

FASCICLE X.7 – Man-Machine Language (MML). Recommendations Z.301-Z.341 (Study Group X).

CONTENTS OF FASCICLE X.4 OF THE BLUE BOOK

Annex F.2 to Recommendation Z.100

REMARK

Due to the specialized nature of the SDL semantics, this Fascicle is published in English only.

REMARQUE

Etant donné la nature très spéciale de la sémantique du LDS, ce fascicule est publié uniquement en anglais.

OBSERVACIÓN

Debido a la naturaleza especializada de la semántica del LED, este fascículo sólo se publica en inglés.

PRELIMINARY NOTES

1 The Questions entrusted to each Study Group for the Study Period 1989-1992 can be found in Contribution No. 1 to that Study Group.

2 In this Fascicle, the expression "Administration" is used for shortness to indicate both a telecommunication Administration and a recognized private operating agency.

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

Contents

FASCICLE X.4

Annex F.2 to Recommendation Z.100

SDL FORMAL DEFINITION
STATIC SEMANTICS

Introduction

This part of The Formal Definition defines the static properties of SDL. For a description of the over-all structure of the Formal Definition and for an explanation of the notation used, refer to Annex F.1: Introduction to the Formal Definition.

Annex F.2 defines the following:

- The well-formedness conditions which apply.
  However, the well-formedness conditions which involve elaboration of equations are, for convenience, deferred to the construction of *Entity-dict* in Annex F.3: Dynamic Semantics (see the introduction to section 5 of Annex F.3). Even though the conditions are placed in the Dynamic Semantics, the conditions are static properties and they are therefore applied before any interpretation of SDL processes takes place.

  Those well-formedness conditions are the following:

  1. The check that the answers in a decision do not overlap.
  2. The check that an enclosed scopeunit does not invalidate the properties of a sort defined in the enclosing scopeunit.
  3. The check that no literal equals the error term.
  4. The check that the Boolean literals TRUE and FALSE denote different values.

- The relation between the concrete syntax and the abstract syntax as defined in Z.100 (referred to as AS_1). AS_1 is not repeated in the Formal Definition. A summary can be found in Annex B of Z.100.

  In order to distinguish the AS_1 objects from other objects in the Meta-IV functions, every AS_1 name is suffixed by a "$_1$".

  For example the definition of *identifier* as defined in §2.2.2 of Z.100 is

  Identifier :: *Qualifier Name*

  whereas the corresponding definition in the Formal Definition is taken as

  $Identifier_1$ :: $Qualifier_1$ $Name_1$

1 Abstract Syntax Representing Concrete Syntax

In the following, the domain AS_0 which reflects the concrete syntax is defined. Although AS_0 presents the concrete syntax on an abstract form, AS_0 does in some respect define the language syntax on a "lower level" than the syntax rules found in Z.100. This is due to the fact that the context sensitive information found in the syntax rules is not reflected in AS_0 (for an explanation see section 3 of Annex F.1)

As opposed to in AS_1, abbreviations for the domain names are used in AS_0 in order to reduce problems of the limited physical line width. The annotations attached to the domain definitions, define the full names, by using *italics* letter style for the letters which also occur in the abbreviations.

In order to distinguish the AS_0 objects from other objects in the Meta-IV functions, every AS_0 domain is suffixed by a "$_0$".

Due to the fact that AS_0 is an Abstract Syntax derived from the textual grammar, the following items are not covered by the Formal Definition:

- Lexical rules (including macros)
  Most of these rules are defined formally in §2.2.1 of Z.100
- Syntax rules (AS_0 does not define where to place keywords and separators, it only defines the objects a system definition consist of).
  The syntax rules are defined formally in Z.100 in the subsections *textual grammar*
- The relation between the textual grammar and the graphical grammar
  In most cases this relation is straigth forward, i.e. <diagram> corresponds to <definition>, "**is followed by**" in the graphical grammar is literally "is followed by" in the textual grammar etc.

The domains are specified in the same order as the corresponding syntax rules are defined in Z.100, e.g. the first domain definition in AS_0 is Id_0 and the first BNF production in Z.100 is <identifier>

1.1 Basic SDL

1	Id_0	::	$Qualifier_0$ $Name_0$
2	$Qualifier_0$	=	$Qualifierelem_0{}^*$
3	$Qualifierelem_0$	=	$Scopeunit_0$ $Name_0$
4	$Scopeunit_0$	=	SYSTEM | BLOCK | PROCESS | SERVICE | PROCEDURE | SUBSTRUCTURE | TYPE | SIGNAL
5	$Name_0$	::	$Char^+$ [EXCLAMATIONMARK]

- An *Id*entifier consists of a *Qualifier* and a *Name*
- A *Qualifier* is a possible empty list of *Qualifier elem*ents.
- A *Qualifier elem*ent is a *scope unit* type and a *Name*.
- A *scope unit* type is either the system, a block, a process, a service, a procedure, a block/channel sub-structure, a partial data type definition or a signal refinement.
- A *Name* consists of a non-empty list of characters followed by an optional exclamationmark.
  For convenience, The *Name*s in AS_0 covers a slightly different class of names than defined by the BNF syntax rules in Z.100.
  The following presumptions on AS_0 apply:
  - The optional exclamation mark following the spelling ($Char^+$) can be present only where it is allowed syntactically according to the BNF rules.

– The spelling is an uppercase sequence of characters which conforms to the lexical rules defined in Z.100.

6 *String*$_0$ :: *Char**

- A character *String* consists of a possible empty list of characters.

7 *Text*$_0$ :: *String*$_0$

- Informal *Text* contains a character *String*. In expression context, informal *Text* cannot be distinguished from *String*, Therefore, in AS$_0$ *Text* only occurs in *Task* as alternative to assignment statements.

8 *Sys*$_0$ :: *Sysdef*$_0$ *Remotedef*$_0$

- A *Sys*tem consists of a *Sys*tem *def*inition and the *Remote def*initions.

9 *Remotedef*$_0$ = *Refdecl*$_0^*$
10 *Refdecl*$_0$ = *Blockdef*$_0$ | *Prdef*$_0$ | *Servicedef*$_0$ | *Procdef*$_0$ | *Chansubdef*$_0$

- The *Remote def*initions is a list of *Ref*erenced *decl*arations.
- A *Ref*erenced *decl*aration is either a *Block def*inition, a *Pr*ocess *def*inition, a *Service def*inition, a *Pro*cedure *def*inition, or a *Chan*nel *sub*structure *def*inition.

Also block sub-structures may be remote. However, as it is not possible to distinguish syntactically between a remote block sub-structure and a remote channel sub-structure, the channel sub-structure must be converted to a block sub-structure (see *remove-references*) if the context shows that the remote definition actually is a block sub-structure.

11 *Sysdef*$_0$ :: *Sysname*$_0$ *Sysdecl*$_0^*$ [*Tailname*$_0$]
12 *Sysname*$_0$ = *Name*$_0$
13 *Sysdecl*$_0$ = *Blockdef*$_0$ | *Blockref*$_0$ | *Chandef*$_0$ | *Sigdef*$_0$ | *Signallistdef*$_0$ | *Datadef*$_0$ | *Select*$_0$

- A *Sys*tem *def*inition consists of a *Sys*tem *name*, a list of *Sys*tem *decl*arations and an optional *tail*ing *name* ending the *Sys*tem *def*inition.
- A *Sys*tem *name* is a *Name*.
- A *Sys*tem *decl*aration is either a *Block def*inition, a *Block ref*erence, a *Chan*nel *def*inition, a *Sig*nal definition, a *Signal list def*inition, a *Data def*inition or a *Select* definition.

14 *Blockref*$_0$ :: *Blockname*$_0$
15 *Blockdef*$_0$ :: *Blockid*$_0$ *Blockdecl*$_0^*$ [*Blocksub*$_0$] [*Tailid*$_0$]
16 *Blockid*$_0$ = *Id*$_0$
17 *Tailid*$_0$ = *Id*$_0$
18 *Blockdecl*$_0$ = *Sigdef*$_0$ | *Prdef*$_0$ | *Prref*$_0$ | *Datadef*$_0$ | *Connect*$_0$ | *Sigroutedef*$_0$ | *Signallistdef*$_0$ | *Select*$_0$
19 *Blockname*$_0$ = *Name*$_0$

- A *Block ref*erence contains a *Block name*.

- A *Block definition* contains a *Block identifier*, a list of *Block declarations*, an optional *Block substructure* definition and an optional *Tailing identifier* ending the *Block definition*.
  Note that it is not possible to distinguish syntactically between *Name* and *Identifier* in the concrete syntax. Consequently, AS_0 contains *Identifiers* at those places where both *Names* and *Identifiers* are allowed.

- A *Block identifier* is an *Identifier*.

- A *Tailing identifier* is an *Identifier*.

- A *Block declaration* is either a *Signal definition*, a *Process definition*, a *Process reference*, a *Data definition*, a channel to signal route *connection*, a *Signal route definition*, a *Signal list definition* or a *Select*.

- A *Block name* is a *Name*.

20	$Prref_0$	::	$Prname_0$ [$Instances_0$]
21	$Prname_0$	=	$Name_0$
22	$Prdef_0$	::	$Prid_0$ [$Instances_0$] $Parm_0{}^*$ [$Inputset_0$] $Prdecl_0{}^*$ $Processbody_0$ [$Tailid_0$]
23	$Prid_0$	=	Id_0
24	$Prdecl_0$	=	$Vardef_0$ | $Viewdef_0$ | $Datadef_0$ | $Sigdef_0$ | $Signallistdef_0$ | $Importdef_0$ | $Procdef_0$ | $Procref_0$ | $Timerdef_0$ | $Select_0$
25	$Procref_0$	::	$Procname_0$
26	$Inputset_0$	::	[$Signallist_0$]
27	$Processbody_0$	=	$Body_0$ | $Decomposition_0$
28	$Parm_0$	::	$Varname_0{}^+$ $Sortid_0$
29	$Instances_0$	::	[$Initial_0$] [$Maximum_0$]
30	$Initial_0$	=	$Expr_0$
31	$Maximum_0$	=	$Expr_0$

- A *Process reference* contains a *Process name* and an optional number of *Instances* specification.

- A *Process Name* is a *Name*.

- A *Process definition* consists of a *Process identifier*, an optional number of *Instances* specification, a list of process formal *Parameters*, an optional valid *Input* signal *set*, a list of *Process declarations*, the *process Body* and an optional *tailing Identifier* ending the *Process definition*.

- A *Process identifier* is an *Identifier*.

- A *Process declaration* is either a *variable definition*, a *View definition*, a *Data definition*, a *Signal definition*, a *Signal list definition*, an *Import definition*, a *Procedure* definition, a *Procedure reference*, a *Timer definition* or a *Select*.

- A *Procedure reference* consists of a *Procedure name*.

- A valid *Input* signal *set* contains a possibly empty (if **nil**) *Signal list*.

- A *Process body* is either a graph *Body* or a service *Decomposition*.

- A process formal *Parameter* consists of a list of *Variable names* and a *Sort identifier*.

- The number of *Instances* specification consists of an optional *Initial* number and an optional *Maximum* number.

- The *Initial* number is a simple *Expression*.

- The *Maximum* number is a simple *Expression*.

32 $Procdef_0$ :: $Procid_0$ $Procparm_0{}^*$ $Procdecl_0{}^*$ $Body_0$ [$Tailid_0$]
33 $Procid_0$ = Id_0
34 $Procdecl_0$ = $Vardef_0$ | $Datadef_0$ | $Procdef_0$ | $Procref_0$ | $Select_0$
35 $Procname_0$ = $Name_0$
36 $Procparm_0$ = $Inoutparm_0$ | $Inparm_0$
37 $Inoutparm_0$ :: $Varname_0{}^+$ $Sortid_0$
38 $Inparm_0$ :: $Varname_0{}^+$ $Sortid_0$

- A *Procedure definition* consists of a *Procedure identifier*, a list of *Procedure formal parameters*, a list of *Procedure declarations*, a procedure *Body* and an optional *Tailing identifier* ending the *Procedure definition*.
- A *Procedure identifier* is an *Identifier*.
- A *Procedure declaration* is either a *variable definition*, *Data definition*, a *Procedure definition*, a *Procedure reference* or a *Select*.
- A *Procedure name* is a *Name*.
- A *Procedure formal parameter* is either a variable *In/Out parameter* or a variable *In parameter*.
- An *In/Out parameter* consist of a list of *Variable Names* and a *Sort identifier*.
- An *In parameter* consist of a list of *Variable Names* and a *Sort identifier*.

39 $Chandef_0$ :: $Channame_0$ $Chanpath_0$ [$Chanpath_0$] [$Chansub_0$] [$Tailname_0$]
40 $Chanpath_0$ :: $Orig_0$ $Dest_0$ $Signallist_0$
41 $Channame_0$ = $Name_0$
42 $Orig_0$ = $Blockid_0$ | ENV
43 $Dest_0$ = $Blockid_0$ | ENV
44 $Tailname_0$ = $Name_0$

- A *Channel definition* consists of a *Channel* name, one or two *Channel communication paths*, an optional *Channel substructure definition* and an optional *tailing name* ending the *Channel definition*. A *Channel communication path* consists of an *Originating* endpoint, a *Destination* endpoint and a *Signal list*.
- An *Originating* endpoint is either a *Block identifier* or the ENVIRONMENT.
- A *Destination* endpoint is either a *Block identifier* or the ENVIRONMENT.
- A *Name* which ends a definition is a *Tailing name*.

45 $Sigroutedef_0$ :: $Sigroutename_0$ $Sigroutepath_0$ [$Sigroutepath_0$]
46 $Sigroutepath_0$ :: $Origin_0$ $Destination_0$ $Signallist_0$
47 $Origin_0$ = Id_0 | ENV
48 $Destination_0$ = Id_0 | ENV
49 $Sigroutename_0$ = $Name_0$

- A *Signal route definition* contains a *Signal route name* and one or two *Signal route communication paths*.
- A *Signal route communication paths* contains an *Originating* process or service, a *Destination* process or service and a *Signal list*.
- An *Originating* process or service is either a *identifier* or the ENVIRONMENT.
- A *Destination process* is either a *Process identifier* or the ENVIRONMENT.
- A *Signal route name* is a *Name*.

50 *Connect*$_0$ :: *Connectpoint*$_0$ *Id*$_0^+$
51 *Connectpoint*$_0$ = (*Id*$_0$ | ENV)

- A *connection* contains a *connection point* and a list of *Id*entifiers to which it is connected. Note that *Connect*$_0$ both denotes channel endpoint connection, channel to route connection, signal route connection and channel connection since these four kind of connections are not distinguished syntactically if they occur in a select definition.
- A *connection point* is an *Id*entifier or, in the case of a channel endpoint connection, ENV.

52 *Sigdef*$_0$ :: *Sigelem*$_0^+$
53 *Sigelem*$_0$ :: *Signame*$_0$ *Sortid*$_0^*$ [*Refinement*$_0$]
54 *Signame*$_0$ = *Name*$_0$
55 *Sortid*$_0$ = *Id*$_0$

- A *Sig*nal definition consists of a list of *Sig*nal *elem*ents.
- A *Sig*nal *elem*ent consists of a *Sig*nal *name*, a list of *Sort id*entifiers and an optional *Refinement* part.
- A *Sig*nal *name* is a *Name*.
- A *Sort id*entifier is an *Id*entifier.

56 *Signallistdef*$_0$ :: *Signallistname*$_0$ *Signallist*$_0$
57 *Signallistname*$_0$ = *Name*$_0$
58 *Signallist*$_0$ = (*Signallistid*$_0$ | *Sigid*$_0$)$^+$
59 *Signallistid*$_0$ :: *Id*$_0$
60 *Sigid*$_0$ = *Id*$_0$

- A *Signal list def*inition consists of a *signal list name* and a *Signal list*.
- A *Signal list name* is a *Name*.
- A *Signal list* is a non-empty list of *Signal list id*entifiers and *Sig*nal *id*entifiers.
- A *Signal list id*entifier contains an *Id*entifier.
- A *Sig*nal *id*entifier is an *Id*entifier.

Note that, as a *Signal list id*entifier is a tree, it is always possible to distinguish between signal list identifiers and signal identifiers (as is also the case in the concrete syntax).

61 *Vardef*$_0$ :: [REVEALED] [EXPORTED] *Vardefelem*$_0^+$
62 *Vardefelem*$_0$ :: *Varname*$_0^+$ *Sortid*$_0$ [*Value*$_0$]
63 *Varname*$_0$ = *Name*$_0$

- A *Var*iable *def*inition consists of an optional REVEALED attribute, an optional EXPORTED attribute and a list of *Var*iable *def*inition *elem*ents.
- A *Var*iable *def*inition *elem*ent consists of a list of *Var*iable *name*s, a *Sort id*entifier and an optional initial *Value*.
- A *Var*iable *name* is a *Name*.

64 *Viewdef*$_0$ :: *Viewdefelem*$_0^+$
65 *Viewdefelem*$_0$ :: *Varid*$_0^+$ *Sortid*$_0$
66 *Varid*$_0$ = *Id*$_0$

- A *View definition* consists of a list of *View definition elements*.
- A *View definition element* consists of a list of view *Variable identifiers* and a *Sort identifier*.
- A *Variable identifier* is an *Identifier*.

67 $Body_0$ :: $Transition_0\ Statebody_0^*$
68 $Statebody_0$ :: $(Statenamelist_0 \mid Starredlist_0)\ Statespec_0^*\ [Tailname_0]$
69 $Statenamelist_0$ :: $Statename_0^+$
70 $Statename_0$ = $Name_0$
71 $Starredlist_0$ :: $[Statenamelist_0]$

- A *Body* consists of the start *Transition* and a list of *State bodys*.
- A *State body* contains a *State name list* or a *Starred* name *list* and followed by a list of *State* stimulus *specifications* and an optional *tailing name* ending the *State body*.
- A *State name list* consists of a list of *State names*.
- A *State name* is a *Name*.
- A *Starred* name *list* contains a possible empty (if **nil**) *State name list*.

72 $Statespec_0$ = $Savespec_0 \mid Inputspec_0 \mid Contspec_0 \mid Priinput_0$

- A *State* stimulus *specification* is either a *Save specification*, an *Input specification*, a *Continuous* signal *specification* or a *Priority input specification*.

73 $Savespec_0$ :: $Signallist_0 \mid Starred_0$
74 $Inputspec_0$ :: $(Starred_0 \mid Inputvars_0^+)\ Enabling_0\ Transition_0$
75 $Starred_0$ :: ()
76 $Inputvars_0$ :: $Sigid_0\ [Varid_0]^*$

- A *Save specification* contains either a *Signal list* or an asterisk.
- An *Input specification* contains either an asterisk (i.e. a tree *Starred* denoting the asterisk) or a none-empty list of *Input* signal *variables* and followed by an *enabling* condition and followed by a *Transition*.
- *Input* signal *variables* contains a *Signal identifier* and a list of optional *Variable identifiers*.

77 $Transition_0$ :: $Actstmt_0^*\ [Termstmt_0]$
78 $Actstmt_0$ :: $[Label_0]\ Act_0$
79 $Label_0$ = $Name_0$

- A *Transition* contains a list of *Action statements* and an optional *Terminator statement*.
  If *Termstmt* is omitted, then the list of *Actstmt* is non-empty and if the list of *Actstmt* is empty then *Termstmt* is present (it need not to be checked as it is a context free concrete syntax rule).
- An *Action statement* contains an optional *Label* and an *Action*.
- A *Label* is a *Name*.

80 Act_0 = $Task_0$ | $Output_0$ | $Create_0$ | $Decision_0$ | $Export_0$ | $Option_0$ | $Call_0$ | $Prioutput_0$ | Set_0 | $Reset_0$

- An *Action* is either a *Task*, an *Output*, a *Create* request, a *Decision*, an *Export*, a transition *Option*, a procedure *Call*, a *Priority output*, a timer *Set* or a timer *Reset*.

81 $Termstmt_0$ :: [$Label_0$] $Terminator_0$
82 $Terminator_0$ = $Nextstate_0$ | $Join_0$ | $Stop_0$ | $Return_0$
83 $Nextstate_0$ :: [$Statename_0$]
84 $Join_0$ :: $Label_0$
85 $Stop_0$ :: ()
86 $Return_0$ :: ()

- A *Terminator statement* contains an optional *Label* and a *Terminator*
- A *Terminator* is either a *Nextstate*, a *Join*, a *Stop* or a *Return*.
- A *Nextstate* contains an optional *State name* (**nil** if it is a dash nextstate).
- A *Join* contains a *Label*
- A *Stop* contains no additional information.
- A *Return* contains no additional information.

87 $Task_0$ :: $Assignstmt_0^+$ | $Text_0^+$

- A *Task* contains either a list of *Assignment statements* or a list of informal *Texts*.

88 $Create_0$ :: $Prid_0$ $Actparmlist_0$
89 $Actparmlist_0$ = $[Expr_0]^*$

- A *Create* request contains a *Process identifier* and an *Actual parameter list*.
- An *Actual parameter list* is a list of optional *Expressions*.

90 $Call_0$ :: $Procid_0$ $Actparmlist_0$

- A procedure *Call* contains a *Procedure identifier* and an *Actual parameter list*.

91 $Output_0$ :: $Outputsig_0^+$ [$Piexpr_0$] Via_0
92 $Outputsig_0$ :: $Sigid_0$ $Actparmlist_0$
93 $Piexpr_0$ = $Expr_0$ | $Scopeexpr_0$
94 Via_0 = Id_0^*
95 $Scopeexpr_0$ :: $Scope_0$ $Expr_0$
96 $Scope_0$ = $Qualifier_0$

- An *Output* contains a list of *Output signals*, an optional *PId expression* and a *Via*.
- An *Output signal* consists of a *Signal identifier* and an *Actual parameter list*.
- A *PId expression* is either an *Expression* or a *scope expression*.
- A *Via* is a possible empty list of *Identifiers*.

- *scopeexpr*ession has no corresponding construct in the concrete syntax. It is a utility construct which is introduced in AS_0 in order to ease the transformation of services.

  When services are transformed, they are merged into one AS_0 process graph. Except for the transition containing output and decisions implied from enabling condition and continuous signals, every transition in the resulting AS_0 process graph must be transformed in the context of a certain service. The *Scope* in *Scopeexpr*ession is a *Qualifier* which points out the service which "owns" the *Expr*ession occurring in the decisions in enabling condition.

  Scopeexpr$_0$s are generated during the expansion of enabling condition and continuous signal.

97	*Decision*$_0$	::	*Question*$_0$ *Answer*$_0^+$ [*Elsepart*$_0$]
98	*Question*$_0$	=	*Expr*$_0$ | *Scopeexpr*$_0$
99	*Elsepart*$_0$	::	[*Transition*$_0$]
100	*Answer*$_0$	::	*Conditionlist*$_0$ [*Transition*$_0$]

- A *Decision* contains a *Question*, a none-empty list of *Answer*s and an optional *Else part*
- A *Question* is either an *Expr*ession or a *scopeexpr* (explained above).
- An *Else part* contains an optional *Transition.*
- An *Answer* contains a *Condition list* and an optional *Transition*

101	*Timerdef*$_0$	::	*Timerelem*$_0^+$
102	*Timerelem*$_0$	::	*Timername*$_0$ *Sortid*$_0^*$
103	*Timername*$_0$	=	*Name*$_0$

- A *Timer def*inition consists of a list of *Timer elem*ents.
- A *Timer elem*ent contains a *Timer name* and a list of *Sort id*entifiers.
- A *Timer name* is a *Name.*

104	*Reset*$_0$	::	*Resetelem*$_0^+$
105	*Resetelem*$_0$	::	*Timerid*$_0$ *Expr*$_0^*$
106	*Timerid*$_0$	=	*Id*$_0$

- A timer *Reset* contains a list of timer *Reset elem*ents.
- A timer *Reset elem*ent contains a *Timer id*entifier and a possible empty list of *Expr*essions.
- A *Timer id*entifier is an *Id*entifier.

107	*Set*$_0$	::	*Setelem*$_0^+$
108	*Setelem*$_0$	::	*Expr*$_0$ *Timerid*$_0$ *Expr*$_0^*$

- A timer *Set* contains a list of timer *Set elem*ents.
- A timer *Set elem*ent contains a time expiration *Expr*ession, a *Timer id*entifier and a possible empty list of *Expr*essions.

1.2 Structural Concepts

1 *Blocksub*$_0$ = *Blocksubdef*$_0$ | *Blocksubref*$_0$
2 *Blocksubdef*$_0$ :: [*Blocksubid*$_0$] *Blocksubdecl*$_0^+$ [*Tailid*$_0$]
3 *Blocksubid*$_0$ = *Id*$_0$
4 *Blocksubdecl*$_0$ = *Signallistdef*$_0$ | *Connect*$_0$ | *Blockdef*$_0$ | *Blockref*$_0$ | *Chandef*$_0$ | *Sigdef*$_0$ | *Datadef*$_0$ | *Select*$_0$
5 *Blocksubref*$_0$ :: *Blocksubname*$_0$
6 *Blocksubname*$_0$ = *Name*$_0$

- A *Block substructure* is either a *Block substructure definition* or a *Block substructure reference*.
- A *Block substructure definition* consists of a *Block substructure identifier*, a list of *Block substructure declarations* and an optional *tailing identifier* ending the *Block substructure definition*.
- A *Block substructure identifier* is an *Identifier*.
- A *Block substructure declaration* is either a *Signal list definition*, a Block sub-structure *Connection*, a *Block definition*, a *Block reference*, a *Channel definition*, a *Signal definition*, a *Data definition* or a *Select*.
- A *Block substructure reference* contains a *Block substructure name*.
- A *Block substructure name* is a *Name*.

7 *Chansub*$_0$ = *Chansubdef*$_0$ | *Chansubref*$_0$
8 *Chansubdef*$_0$ :: *Chansubid*$_0$ *Chansubdecl*$_0^+$ [*Tailid*$_0$]
9 *Chansubid*$_0$ = [*Id*$_0$]
10 *Chansubdecl*$_0$ = *Chandef*$_0$ | *Blockdef*$_0$ | *Sigdef*$_0$ | *Blockref*$_0$ *Signallistdef*$_0$ | *Datadef*$_0$ | *Connect*$_0$ | *Select*$_0$
11 *Chansubref*$_0$ :: *Chansubname*$_0$
12 *Chansubname*$_0$ = *Name*$_0$

- A *Channel substructure* is either a *Channel substructure definition* or a *Channel substructure reference*.
- A *Channel substructure definition* consists of a *Channel substructure identifier*, a list of *Channel substructure declarations* and an optional *Tailing identifier* ending the *Channel substructure definition*.
- A *Channel substructure identifier* is an optional *Identifier*.
- A *Channel substructure declaration* is either a *Channel definition*, a *Block definition*, a *Signal definition*, a *Block reference*, a *Signal list definition*, a *Data definition*, a channel endpoint *connection* or a *Select*.
- A *Channel substructure reference* contains a *Channel substructure name*.
- A *Channel substructure name* is a *Name*.

13 *Refinement*$_0$ :: *Subsignal*$_0^+$
14 *Subsignal*$_0$ :: [REVERSE] *Sigdef*$_0$

- A *Refinement* part contains a list of *Subsignal* definitions.
- A *Subsignal* definition contains a flag indicating whether the subsignal is leading in the opposite direction and a *Signal* definition.

1.3 Additional Concepts

1 *Select*$_0$ :: *Expr*$_0$ *Decl*$_0^+$
2 *Decl*$_0$ = *Blockdef*$_0$ | *Blockref*$_0$ | *Chandef*$_0$ | *Sigdef*$_0$ | *Prdef*$_0$ | *Prref*$_0$ | *Procdef*$_0$ | *Procref*$_0$ | *Servicedef*$_0$ | *Serviceref*$_0$ | *Datadef*$_0$ | *Sigroutedef*$_0$ | *Signallistdef*$_0$ | *Vardef*$_0$ | *Importdef*$_0$ | *Viewdef*$_0$ | *Timerdef*$_0$ | *Connect*$_0$ | *Select*$_0$

- A *Select* definition consists of an *Expr*ession and a list of *Decl*arations.
- A *Decl*aration is either a *Block def*inition, a *Block ref*erence, a *Chan*nel *def*inition, a *Sig*nal *def*inition a *Pro*cess *def*inition, a *Pro*cess *ref*erence, a *Pro*cedure *def*inition, a *Pro*cedure *ref*erence, a *Service def*inition, a *Service ref*erence, a *Data def*inition, a *Sig*nal *route def*inition, a *Signal list def*inition, a *Var*iable *def*inition, an *Import def*inition, a *View def*inition,a *Timer def*inition, a *connec*tion or a *Select* definition.

3 *Option*$_0$ :: *Question*$_0$ *Answer*$_0^+$ [*Elsepart*$_0$]

- A transition *Option* contains a *Question*, a list of *Answer*s and an optional *Else part.*

4 *Decomposition*$_0$ :: *Decompositiondecl*$_0^+$
5 *Decompositiondecl*$_0$ = *Sigroutedef*$_0$ | *Connect*$_0$ | *Servicedef*$_0$ | *Serviceref*$_0$ | *Select*$_0$

- A *Decomposition* contains a list of *Decomposition decl*arations.
- A *Decomposition decl*aration is either a *Sig*nal *route def*inition, a signal route *con*nection, a *Service def*inition a *Service ref*erence or a *Select* definition.

6 *Servicedef*$_0$ :: *Serviceid*$_0$ [*Inputset*$_0$] *Servicedecl*$_0^*$ *Body*$_0$ [*Tailid*$_0$]
7 *Servicedecl*$_0$ = *Vardef*$_0$ | *Procdef*$_0$ | *Procref*$_0$ | *Viewdef*$_0$ | *Importdef*$_0$ | *Timerdef*$_0$ | *Datadef*$_0$ | *Select*$_0$
8 *Serviceid*$_0$ = *Id*$_0$
9 *Serviceref*$_0$ :: *Servicename*$_0$
10 *Servicename*$_0$ = *Name*$_0$

- A *Service def*inition consists of a *Service id*entifier, an optional valid service *input* signal *set*, a list of *Service decl*arations, a service *Body* and an optional *Tail*ing *id*entifier ending the *Service def*inition.
- A *Service decl*aration is either a *Var*iable *def*inition, a *pro*cedure *def*inition, a *Proce*dure *ref*erence, a *View def*inition, an *Import def*inition, a *Timer def*inition, a *Data def*inition or a *Select* definition.
- A *Service id*entifier is an *Id*entifier.
- A *Service ref*erence contains a *Service name.*
- A *Service name* is a *Name.*

11 *Priinput*$_0$ :: *Inputvars*$_0^+$ *Transition*$_0$
12 *Prioutput*$_0$ :: *Outputsig*$_0^+$

- A *Pri*ority *input* consists of a list of signal *input var*iables and a *Transition.*
- A *Pri*ority *output* contains a list of *Output sig*nals.

13 *Contspec*$_0$ :: *Expr*$_0$ *Priority*$_0$ *Transition*$_0$
14 *Priority*$_0$ = [*Name*$_0$]
15 *Enabling*$_0$ = [*Expr*$_0$]

- A *Con*tinuous signal *spec*ification contains an *Exp*ression, a *Priority* and a *Transition*
- *Priority* is an optional integer *Name*
- *Enabling* is an optional *Exp*ression.

16 *Export*$_0$ :: *Varid*$_0^+$

- An *Export* contains a list of exported *Var*iable *id*entifiers.

17 *Importdef*$_0$ :: *Importelem*$_0^+$
18 *Importelem*$_0$ :: *Varname*$_0^+$ *Sortid*$_0$
19 *Importexpr*$_0$ :: *Varid*$_0$ [*Expr*$_0$]

- An *Import def*inition contains a list of *Import elem*ents.
- An *Import elem*ent is a list of imported *Var*iable *names* and a *Sort id*entifier.
- An *Import exp*ression contains an imported *Var*iable *id*entifier and an optional PId *Exp*ression.

1.4 Data

1 *Partialtypedef*$_0$ :: *Sortname*$_0$ [*Extproperties*$_0$] *Properties*$_0$ [*Conditionlist*$_0$] [*Tailname*$_0$]
2 *Sortname*$_0$ = *Name*$_0$

- A *Partial type def*inition consists of a *Sort name*, some optional *Ext*ended *properties*, some *Properties*, an optional range *condition* value *list* and an optional sort *name* ending the definition.
  Note that, as opposed to the syntax rule in Z.100, this definition also covers syntypes where the partial type definition is implied (i.e. in the case where *Conditionlist*$_0$ is different from **nil**).
- A *Sort name* is a *Name*.

3 *Properties*$_0$ :: *Literal*$_0^*$ *Op*$_0^*$ *Axiom*$_0^*$ *Mappingaxiom*$_0^*$ [*Initialvalue*$_0$]

- *Properties* consists of a list of *Literal* signatures, a list of *Op*erator signatures, a list of *Axioms*, a list of literal *Mapping axioms* and an optional *Initial* (default) *value*.

4 *Literal*$_0$ = *Name*$_0$ | *String*$_0$ | *Nmclass*$_0$

- A *Literal* signature is either a *Name*, a character *String* or a *Nameclass*.

5 *Op*$_0$ = *Ordering*$_0$ | *Opspec*$_0$
6 *Opspec*$_0$ :: *Operatorname*$_0$ *Sortid*$_0^+$ *Sortid*$_0$
7 *Operatorname*$_0$ = *Name*$_0$ | *Quotedop*$_0$

- An *Operator* signature is either *Ordering* or an *Operator specification*.

- An *Operator specification* consists of an *Operator name*, a list of argument *Sort identifiers* and a result *Sort identifier*.

- An *Operator name* is a *Name* or a *Quoted operator*.

8 $Axiom_0$ = $Unquantequation_0$ | $Condequation_0$ | $Quantequation_0$
9 $Unquantequation_0$ = $Equation_0$ | $Term_0$

- An *Axiom* is either an *Unquantified Equation* or a *Conditional equation* or a *Quantified equation* or a *Term*.

- An *Unquantified Equation* is either an *Equation* or a *Term*.

10 $Equation_0$ :: $Term_0$ $Term_0$

- An *Equation* consists of a left-hand *Term* and a right-hand *Term*.

11 $Quantequation_0$ :: $Valuename_0^+$ $Sortid_0$ $Axiom_0^+$
12 $Valuename_0$ = $Name_0$

- A *Quantified equation* consists of a list of *Value names*, a *Sort identifier* and a list of *Axioms*.

- A *Value name* is a *Name*.

13 $Term_0$ = Id_0 | $Operatorterm_0$ | $Condterm_0$ | $Stringterm_0$
$Monadterm_0$ | $Infixterm_0$ | $Errorterm_0$ | $Spellingterm_0$

- A *Term* is either an *Identifier*, an *Operator term*, a *Conditional term*, a character *String term*, a *Monadic term*, an *Infix term*, an *Error term* or a *Spelling term*.
Note that, in AS_0, no distinction can be made between the various kinds of operators in terms and expressions which only differs semantically. For example, ⟨extended composite term⟩ and ⟨extended ground term⟩ as defined in §5.4 of Z.100, cannot be distinguished syntactically (i.e. without binding names to definitions), which means that they are covered by the same domain definition in AS_0.

14 $Operatorterm_0$ :: (Id_0 | $Qualop_0$) $Term_0^+$

- An *Operator term* consists of an *Identifier* or a *Qualified operator* and followed by a list of argument *Terms*.

15 $Condterm_0$ :: $Term_0$ $Term_0$ $Term_0$

- A *Conditional term* consists of a condition *Term*, a consequence *Term* and an alternative *Term*.

16 $Stringterm_0$ :: $Qualifier_0$ $String_0$

- A character *String term* consists of a *Qualifier* and a character *string*.

17 *Monadterm*$_0$ :: (NOT | MINUS) *Term*$_0$

- A *Monadic term* consists of one of the operators "NOT" or "-" followed by an argument *Term*

18 *Infixterm*$_0$ :: *Term*$_0$ *Infixop*$_0$ *Term*$_0$
19 *Infixop*$_0$ = IMPLY | OR | XOR | AND | IN | MOD | REM | PLUS | MINUS | CONC | MULT | DIV | *Relop*$_0$
20 *Relop*$_0$ = NE | EQ | GT | LT | LE | GE

- An *Infix term* consists of two argument *Terms* and an *Infix operator*.
- An *Infix operator* is either "=)","OR","XOR","AND","IN", "MOD","REM","+","-","//","*","/ or one of the *Relational operators*.
- A *Relational operator* is either "/=","=",">","<","<=" or ">=".

Note that the grouping of any sub-trees reflects the precedence rules as defined in §5.4.2.2 of Z.100.

21 *Errorterm*$_0$ :: ()

- An *Error term* contains no additional information.

22 *Condequation*$_0$ :: *Unquantequation*$_0$$^+$ *Unquantequation*$_0$

- A *Conditional equation* consists of a list of restriction *Unquantified Equations* and a restricted *Unquantified Equation*.

23 *Extproperties*$_0$ = *Struc*$_0$ | *Inherited*$_0$ | *Geninstlist*$_0$

- *Extended properties* is either *Structure* properties, *Inherited* properties or *Generator instance list* properties.

24 *Syntypedef*$_0$ :: *Syntypename*$_0$ *Parentid*$_0$ [*Initialvalue*$_0$] [*Conditionlist*$_0$] [*Tailname*$_0$]
25 *Syntypename*$_0$ = *Name*$_0$

- A *Syntype definition* consists of a *Syntype name*, a *Parent identifier*, an optional *Initial value* an optional range *Condition list* and a syntype name ending the definition.
- A *Syntype name* is a *Name*.

26 *Conditionlist*$_0$ = *Condition*$_0$$^+$
27 *Condition*$_0$ :: [*Value*$_0$ | *Relop*$_0$] *Value*$_0$
28 *Value*$_0$ = *Expr*$_0$

- A range *Condition list* is a list of range *Conditions*
- A range *Condition* consists of an optional *Value* or *Relational operator* and followed by a *Value*.
- A *Value* is an *Expression*.

29 $Struc_0$ :: $Fieldspec_0{}^+$
30 $Fieldspec_0$ :: $Fieldname_0{}^+$ $Sortid_0$
31 $Fieldname_0$ = $Name_0$

- *Struc*ture properties consists of a list of *Field spec*ifications.
- A *Field spec*ification consists of a list of *Field name*s and a *Sort id*entifier.
- A *Field name* is a *Name*

32 $Inherited_0$ :: $Parentid_0$ $Literalrenaming_0$ (ALL | $Operatorrenaming_0$)
33 $Parentid_0$ = Id_0
34 $Literalrenaming_0$ = $Literalpair_0{}^*$
35 $Literalpair_0$ :: $Newliteral_0$ $Oldliteral_0$
36 $Newliteral_0$ = $Name_0$ | $String_0$
37 $Oldliteral_0$ = $Name_0$ | $String_0$
38 $Operatorrenaming_0$ = $Operatorpair_0{}^*$
39 $Operatorpair_0$ :: $Newoperator_0$ $Oldoperator_0$
40 $Newoperator_0$ = [$Operatorname_0$]
41 $Oldoperator_0$ = $Operatorname_0$

- *Inherited* properties consists of a *Parent id*entifier, *Literal renaming* and inherited operators which are either ALL operators or *Operator renaming.*
- A *Parent id*entifier is an *Id*entifier.
- *Literal renaming* is a list of *Literal pair*s.
- A *Literal pair* consists of the *New literal* name and the *Old* (parent) *Literal* name.
- The *New literal* name is either a *Name* or a character *String.*
- The *Old literal* name is either a *Name* or a character *String.*
- *Operator renaming* is a list of *Operator pair*s.
- An *Operator pair* consists of the *New operator* name and the *Old operator* name.
- The *New operator* name is an *Operator name* if specified.
- The *Old* (parent) *operator* name is an *Operator name.*

42 $Sortgenerator_0$ :: $Generatorname_0$ $Genparm_0{}^+$
[$Geninstlist_0$] $Properties_0$ [$Tailname_0$]
43 $Generatorname_0$ = $Name_0$
44 $Genparm_0$ = $Sortparm_0$ | $Termparm_0$ | $Litparm_0$ | $Opparm_0$
45 $Sortparm_0$ :: $Name_0{}^+$
46 $Termparm_0$ :: $Name_0{}^+$
47 $Litparm_0$ :: $Name_0{}^+$
48 $Opparm_0$ :: $Name_0{}^+$

- A *Sort generator* consists of a *Generator name*, a list of *Gen*erator formal *para*meters, a possible empty *Generator inst*ance *list*, some *Properties* and a Sort generator *Name* ending the definition.
- A *Generator name* is a *Name.*
- A *Gen*erator formal *para*meter is either a *Sort para*meter, a *Term para*meter, a *Lit*eral *para*meter or an *Op*erator *para*meter.
- A *Sort para*meter consists of a list of *Name*s.

- A *Term parameter* consists of a list of *Names*.
- A *Literal parameter* consists of a list of *Names*.
- An *Operator parameter* consists of a list of *Names*.

49	$Geninstlist_0$	::	$Geninst_0{}^+$
50	$Geninst_0$	::	$Generatorid_0\ Genactparm_0{}^+$
51	$Generatorid_0$	=	Id_0
52	$Genactparm_0$	=	$Term_0 \mid Quotedop_0 \mid Nmclass_0$

- A *Generator instance list* consists of a list of *Generator instances*.
- A *Generator instance* consists of a *Generator identifier* and a list of *Generator actual parameters*.
- A *Generator identifier* is an *Identifier*.
- A *Generator actual parameter* is a *Term* or an *Quoted operator* or a *Name class*. I.e. *Sort identifiers* and *Operator names* are *Terms* syntactically, like other kinds of actual parameters. *Quoted operators* and *Name class*, on the other hand, cannot form a *Term* on its own and are therefore specified explicit as alternatives in $Genactparm_0$.

53	$Synonymdef_0$	::	$Synonymname_0\ [Sortid_0]\ [Initialvalue_0]$
54	$Synonymname_0$	=	$Name_0$

- A *Synonym definition* consists of a *Synonym name* an optional *Sort identifier* and an optional *Initial* (default) *value*.
- A *Synonym name* is a *Name*.

55	$Nmclass_0$	::	$Regularexp_0$
56	$Regularexp_0$	=	$Partregexp_0 \mid Orregexp_0 \mid Andregexp_0$
57	$Orregexp_0$	::	$Regularexp_0\ Partregexp_0$
58	$Andregexp_0$	::	$Regularexp_0\ Partregexp_0$
59	$Partregexp_0$	=	$Rngregexp_0 \mid Singregexp_0 \mid Parenregexp_0$
60	$Rngregexp_0$	::	$String_0\ String_0\ [Regexpexp_0]$
61	$Singregexp_0$	::	$String_0\ [Regexpexp_0]$
62	$Parenregexp_0$	::	$Regularexp_0\ [Regexpexp_0]$
63	$Regexpexp_0$	=	MULT | PLUS | $Name_0$

- A *Nameclass* consists of a *Regular expression*.
- A *Regular expression* is either a *Partial regular expression* or an infix *Or regular expression* or an infix *And regular expression*.
- An infix *Or regular expression* consists of a *Regular expression* and a *Partial regular expression*.
- An infix *And regular expression* consists of a *Regular expression* and a *Partial regular expression*.
- A *Partial Regular expression* is either a *Range regular expression* or a *single regular expression* or a *Parenthesis regular expression*.
- A *Range regular expression* consists of two character *Strings* and an optional repetition *regular expression Expression*.
- A *Single regular expression* consists of a character string and an optional repetition *regular expression Expression*.

- A *Paren*thesis *reg*ular *ex*pression consists of a *reg*ular *ex*pression and an optional repetition *reg*ular *ex*pression *Ex*pression.
- A repetition *reg*ular *ex*pression *Ex*pression is either a "*" or a "+" or *Name*.

64 *Ordering*$_0$ :: ()

- *Ordering* contains no additional information.

65 *Mappingaxiom*$_0$ :: *Valuename*$_0^+$ *Sortid*$_0$ *Literalaxiom*$_0^+$
66 *Literalaxiom*$_0$ = *Axiom*$_0$ | *Mappingaxiom*$_0$
67 *Spellingterm*$_0$ :: *Id*$_0$

- A *Mapping axiom* consists of a list of *Value name*s, a *Sort id*entifier and a list of *Literal axiom*s.
- A *Literal axiom* is either an *Axiom* or a *Mapping axiom*.
- A *Spelling term* contains an *Id*entifier.

68 *Expr*$_0$ = *Id*$_0$ | *Stringterm*$_0$ | *Condexpr*$_0$ | *Operatorapp*$_0$ | *Monadexpr*$_0$ | *Infixexpr*$_0$ | *Impoperator*$_0$ | *Selectexpr*$_0$ | *Tupleexpr*$_0$

- An *Ex*pression is either an *Id*entifier or a character *String term* or a *Cond*itional *ex*pression or an *Operator ap*plication or a *Monadic ex*pression or an *Infix ex*pression or an *Imp*erative operator or a field *Select ex*pression or a structure *tuple ex*pression.

69 *Monadexpr*$_0$ :: (NOT | MINUS) *Expr*$_0$

- A *Monad*ic *Ex*pression consists of one of the operators "NOT","-" and followed by an argument *Ex*pression.

70 *Infixexpr*$_0$ :: *Expr*$_0$ *Infixop*$_0$ *Expr*$_0$

- An *Infix ex*pression consists of two argument *Ex*pressions and an *Infix o*perator.

71 *Selectexpr*$_0$ :: *Expr*$_0$ *Name*$_0$

- A field *Select ex*pression consists of an *Ex*pression and a field *Name*.

72 *Tupleexpr*$_0$ :: *Qualifier*$_0$ *Expr*$_0^+$

- A *Tuple ex*pression contains a *Qualifier* and a list of *Ex*pressions.

73 *Condexpr*$_0$ :: *Expr*$_0$ *Expr*$_0$ *Expr*$_0$

- A *Cond*itional *ex*pression consists of a condition *Ex*pression, a consequence *Ex*pression and an alternative *Ex*pression.

74 *Datadef*$_0$ = *Partialtypedef*$_0$ | *Syntypedef*$_0$ | *Sortgenerator*$_0$ | *Synonymdef*$_0$

- A *Data definition* is either a *Partial type definition*, a *Syntype definition*, a *Sort generator* or a *Synonym definition*.

75 *Exprlist*$_0$ = *Expr*$_0$*

- An *Expression list* is a list of *Expressions*.

76 *Operatorapp*$_0$ :: (*Expr*$_0$ | *Qualop*$_0$) *Exprlist*$_0$
77 *Qualop*$_0$ :: *Qualifier*$_0$ *Quotedop*$_0$
78 *Quotedop*$_0$ :: *Infixop*$_0$ | NOT

- An *Operator application* contains an *Expression* or a *Qualified operator* and followed by an *Expression list*. *Expression* in *Operatorapp* can be either an *Identifier* (denoting an operator *Identifier*) or represent a ⟨primary⟩ as defined in §5.4.2.4 and §5.4.2.5 in Z.100.
- A *Qualified operator* contains a *Qualifier* and a *Quoted* operator.
- A *Quoted operator* is an *Infix operator* or "NOT".

79 *Assignstmt*$_0$ :: *Variable*$_0$ *Expr*$_0$
80 *Variable*$_0$ = *Varid*$_0$ | *Indexedvar*$_0$ | *Fieldvar*$_0$
81 *Indexedvar*$_0$ :: *Variable*$_0$ *Exprlist*$_0$
82 *Fieldvar*$_0$ :: *Variable*$_0$ *Name*$_0$

- An *Assignment statement* contains a *Variable* and an *Expression*.
- A *Variable* is either a *Variable identifier* or an *Indexed variable* or a *Field variable*.
- An *Indexed variable* consists of a *Variable* and an *Expression list*. Note that an *Indexed variable* may denote a *Field variable*, e.g.

```
v(a) := ...
```

may (depending of the context) be another way of writing

```
v!a := ...
```

- A *Field variable* consists of a *Variable* and a field *Name*.

83 *Viewexpr*$_0$ :: *Varid*$_0$ *Expr*$_0$

- A *View expression* consists of a view *variable identifier* and an *Expression*.

84 *Initialvalue*$_0$ = *Expr*$_0$

- An *Initial* (default) *Value* is an *Expression*.

85 *Impoperator*$_0$ = *Importexpr*$_0$ | *Viewexpr*$_0$ | *Nowexpr*$_0$ | *Activeexpr*$_0$ | *Parentexpr*$_0$ | *Offspringexpr*$_0$ | *Senderexpr*$_0$ | *Selfexpr*$_0$
86 *Nowexpr*$_0$ :: ()
87 *Selfexpr*$_0$ :: ()
88 *Parentexpr*$_0$ :: ()
89 *Offspringexpr*$_0$ :: ()
90 *Senderexpr*$_0$ :: ()
91 *Activeexpr*$_0$ :: *Timerid*$_0$ *Exprlist*$_0$

- An *Imperative operator* is either an *Import expression*, a *View expression*, a *Now expression*, a timer *Active expression*, a *Parent expression*, an *Offspring expression*, a *Sender expression* or a *Self expression*.
- The *Now expression* contains no additional information
- The *Self expression* contains no additional information
- The *Parent expression* contains no additional information
- The *Offspring expression* contains no additional information
- The *Sender expression* contains no additional information
- The timer *Active expression* contains a timer *identifier* and an *Expression list*

2 Internal Domains

The semantic domains define the domain of the composite object which holds some derived (context) information (attributes) required during the transformation. There is a distinction between the information attached to given entities, such as information about endpoints for channels, sort for variables, sort list for signals etc., and "common" data such as information about which scopeunit surrounds a definition list, collection of names for implicit variables etc. The two kinds of information (*Descriptordict* and *Quotdict*) are assembled into one object *Dict* such that only one extra formal parameter or one extra result is required for the functions using it or generating it respectively.

1 *Dict* = *Descriptordict* ∪ *Quotdict*

The *Dict* object when given as formal parameter is conventionally named *dict* and appearing in the second argument list in the function headings.

In the following the two domains are described in detail.

2.1 Description of *Descriptordict*

Descriptordict is a mapping of identifiers into their descriptors.

1 *Descriptordict* = *Qual* $\overrightarrow{m}$ *Descr*
2 *Qual* = (*Qualelem* | *Operatorqualelem* | *Importqualelem* | *Viewqualelem*)$^+$

Qual denotes the internal representation of AS_0 identifiers. In SDL, only AS_0 identifiers within the same entity class are unique, so in order to incorporate all identifiers in the same map, the entity class must be part of the map entries. Hence, *Qual* consist of a list of qualifier elements, the final qualifier element in the list being the entity class and the entity name. *Operatorqualelem, Importqualelem* and *Viewqualelem* are treated in a special way because additional information are required in order to guarantee uniqueness for those entities.

example:

A block BL in a system SYS has a *Qual* in *Descriptordict* which is

⟨(SYSTEM, mk-*Name*$_0$("SYS", nil)), (BLOCK, mk-*Name*$_0$("BL", nil))⟩

A signal SIG in block BL has a *Qual* which is

⟨(SYSTEM, mk-*Name*$_0$("SYS", nil)), (BLOCK, mk-*Name*$_0$("BL", nil)), (SIGNAL, mk-*Name*$_0$("SIG", nil))⟩

3 *Qualelem* = *Kind* (*Name*$_0$ | *String*$_0$)
4 *Kind* = *Scopeunit*$_0$ | *Entity*
5 *Entity* = SIGNALROUTE | CHANNEL | SIGNALLIST | GENERATOR | VALUE | LITERAL
6 *Operatorqualelem* = OPERATOR ((*Name*$_0$ | *Quotedop*$_0$) *Sortqual*$^+$ *Sortqual*)
7 *Importqualelem* = IMPORT (*Name*$_0$ *Sortqual*)
8 *Viewqualelem* = VIEW *Qual*

A *Qualelem* is a pair of type of *Scopeunit* and name. When it appears as the final *Qualelem* in the list (in *Qual*) it denotes the entity class and entity name. A *Kind* which is an *Entity* always denotes an entity class, i.e. it always appears as the final *Qualelem* in a *Qual*. This is always the case with the special *Qualelems Operatorqualelem,Importqualelem* and *Viewqualelem*. The entity class VALUE denotes variables, synonyms and value identifiers.

The information (i.e. *Operatorqualelem*) which makes an operator unique (within the "partial data type" scopeunit) is the name (*Name*$_0$) or quoted operator (*Quotedop*$_0$), the argument sorts (*Sortqual*$^+$) and the result sort (*Sortqual*).

The information (i.e. *Importqualelem*) which makes an imported entity unique within a scopeunit is the name (*Name*$_0$) of the imported entity together with its sort (*Sortqual*).

The information (i.e. *Viewqualelem*) which makes a viewed variable unique in a process definition is the identifier (*Qual*) of the viewed variable.

9 *Descr* = *SystemD* | *BlockD* | *ChannelD* | *SignalD* | *TimerD* | *SignalrouteD* | *SignallistD* | *ProcedureD* | *ProcessD* | *SortD* | *SyntypeD* | *GeneratorD* | *SynD* | *VarD* | *ImportD* | *ViewD* | *LiteralD* | *OperatorD* | *ValueidD* |

ServiceD | *BlocksubD* | *ChannelsubD* | *ErrorD*

A *Descr* in the *Descriptordict* is a descriptor of either a system (or the outermost level), a block, a channel, a signal, a timer, a signal route, a signal list, a procedure, a process, a sort, a syntype, a generator, a synonym, a variable, an import variable, a view variable, a literal, an operator, an axiom variable (a value identifier), a service, a block substructure, a channel substructure or a recursive descriptor.

The following Meta-IV assertion on the relation between *Qual*s and their associated *Descr* in *Dict* always applies:

is-consistent-Dict(*dict*) $\triangleq$ (2.1.1)

```
(∀qual ∈ dom dict)
 ((let (kind, ) = qual[len qual] in
   cases kind:
    (SYSTEM
       → is-SystemD(dict(qual)),
     BLOCK
       → is-BlockD(dict(qual)),
     CHANNEL
       → is-ChannelD(dict(qual)),
     SIGNAL
       → is-SignalD(dict(qual)) ∨ is-TimerD(dict(qual)),
     SIGNALROUTE
       → is-SignalrouteD(dict(qual)),
     SIGNALLIST
       → is-SignallistD(dict(qual)) ∨ is-ErrorD(dict(qual)),
     PROCEDURE
       → is-ProcedureD(dict(qual)),
     PROCESS
       → is-ProcessD(dict(qual)),
     TYPE
       → is-SortD(dict(qual)) ∨ is-SyntypeD(dict(qual)),
     VALUE
       → is-VarD(dict(qual)) ∨ is-SynD(dict(qual)) ∨
         is-ValueidD(dict(qual)) ∨ is-ErrorD(dict(qual)),
     GENERATOR
       → is-GeneratorD(dict(qual)) ∨ is-ErrorD(dict(qual)),
     SUBSTRUCTURE
       → is-BlocksubD(dict(qual)) ∨ is-ChannelsubD(dict(qual)),
     LITERAL
       → is-LiteralD(dict(qual)),
     SERVICE
       → is-ServiceD(dict(qual)),
     OPERATOR
       → is-OperatorD(dict(qual)),
     IMPORT
       → is-ImportD(dict(qual)),
     VIEW
       → is-ViewD(dict(qual)))))
```

type: *Dict* → *Bool*

The above function is specified for explanatory reasons only.

10 *SystemD* :: ()
11 *BlocksubD* :: ()

SystemD is a descriptor of the system level.
BlocksubD is a descriptor of a block substructure. These descriptors are only present because the system level and block sub-structures are scopeunits defining entities, and therefore these names are used in the qualifiers in identifiers. (All fully specified qualifiers can be found in the domain of *Dict* as scopeunits.)

12 *ChannelsubD* :: *Blockqual*

ChannelsubD is a descriptor of a channel substructure.

Blockqual In AS_1, a channel substructure is represented as the block substructure of a synthetic block definition. *Blockqual* represents the identifier of this block substructure. Each time an AS_0 identifier referencing a channel substructure in its qualifying part is used, the qualifier is modified in accordance with *Blockqual.*

13 *ServiceD* :: $Transition_0$ *Statedict Labeldict Validinputset Priinputset*
14 *Priinputset* = *Signalqual*-set

ServiceD Is a descriptor of an SDL service.

$Transition_0$ Is the initial transition taken from *Body* of the *Service definition.*

Statedict, Labeldict Is *Statedict* and *Labeldict* for the service. (See the description of *Quotdict*).

Validinputset The complete valid input signal set for the service (including timers).

Priinputset The set of priority input signals for the service.

15 *BlockD* :: *Exportchannels Importchannels Explicitroutes BlockconnectionD*
16 *Exportchannels* = *ExpimpchanD*
17 *Importchannels* = *ExpimpchanD*
18 *ExpimpchanD* = *NameclosureD* $\overrightarrow{m}$(*Otherend Channame*$_0$)*
19 *Otherend* = *Blockqual* | ENV
20 *Explicitroutes* = *Bool*
21 *BlockconnectionD* = *Channelqual* $\overrightarrow{m}$*Qual*-set
22 *Channelqual* = *Qual*

BlockD is a descriptor of a block.

Exportchannels Contains information about the implicit channels leading to and from the block because of export variables in the processes contained in the block.

Importchannels Contains information of the implicit channels leading to and from the block because of import variables in the processes contained in the block. For the implicit channels contained in *importchannels* the information is deduced from *Exportchannels* occurring in other block descriptors.

ExpimpchanD Is a map of the import - export closures (see the domain definition of *NameclosureD*) into the other endpoint of the channel (*Otherend*) and the bidirectional channel which is attached to the closure. The first

signal list in the channel contains the signal xtQUERY and the second signal list in the channel contains the signal xtREPLY.

Explicitroutes — True if explicit signal routes are specified for the block. *Explicitroutes* is used for deriving the complete valid input signal set for the contained processes.

BlockconnectionD — Is the relation between the channels and the signal routes connected to the channel. *BlockconnectionD* is used for replacing channel identifiers by signal route identifiers in VIA constructs of the enclosed processes and for replacing channel identifiers having a sub-structure by the appropriate new channel identifiers in connections.

Channelqual — Is the *Qual* of a channel.

23 *ChannelD* :: *Endpoint Endpoint Signalqual*-**set** *Signalqual*-**set** [*Newchannels*]
24 *Endpoint* = *Blockqual* | **ENV**
25 *Signalqual* = *Qual*
26 *Blockqual* = *Qual*
27 *Newchannels* = *Endpoint* $Channame_0$ *Endpoint* $Channame_0$

ChannelD is a descriptor of a channel.

Endpoint — Is an endpoint which is either a block identifier or the environment.

Signalqual-**set** — Is the complete set of signal identifiers conveyed from *Origin* to *Destination.* In the case of a bidirectional channel, another set of signals is present, denoting the signals conveyed from the second *Endpoint* block to the first *Endpoint* block.

Newchannels — If the channel contains a channel substructure, the channel is represented by two channels in AS_1. *Newchannels* contains the names and originating endpoints of those two channels and the information is used when an old channel identifier is to be replaced by a new channel identifier in a VIA set.

28 *SignalD* :: *Sortqual** *Signalqual*-**set** *Signalqual*-**set**
29 *Sortqual* = *Qual*

SignalD is a descriptor of a signal.

*Sortqual** — is the sorts of values conveyed by the signal.

Signalqual-**set** — The two sets of sub-signals, the first one being the signals leading in the same direction as the parent signal, and the second one being the signals leading in the opposite direction.

30 *TimerD* :: *Sortqual** *Newqual*

TimerD is a descriptor of a timer.

*Sortqual** — is the sorts of the values conveyed by the timer.

Newqual — Denotes the identifier to be used for AS_1. Its name in *Newqual* has changed if the timer is defined in a service.

31	*SignalrouteD*	::	*Originprocess Destinationprocess Signalqual*-**set** *Signalqual*-**set**
32	*Originprocess*	=	*Processqual* | ENV
33	*Destinationprocess*	=	*Processqual* | ENV

SignalrouteD is a descriptor of a signal route.

Originprocess Is the originating endpoint which is either a process identifier or the environment.

Destinationprocess Is the terminating endpoint which is either a process identifier or the environment.

Signalqual-**set** Is the set of signal identifiers conveyed from *Origin* to *Destination.* In the case of a bidirectional signal route, another set of signals is present, denoting the signals conveyed from *Destination* process to *Origin* process.

34	*SignallistD*	::	$Signallist_0$

A *SignallistD* is a descriptor of a signallist. It contains the list of AS_0 signal identifiers (and signal list identifiers) attached to the signal list.

35	*ProcedureD*	::	*FormparmD* Newqual*
36	*FormparmD*	=	*InDescr* | *InoutDescr*
37	*InDescr*	::	*Sortqual*
38	*InoutDescr*	::	*Sortqual*
39	*Newqual*	=	*Qual*

ProcedureD is a descriptor of a procedure.

*FormparmD** A list of special descriptors used when the procedure is invoked, to check for sort compability of the actual parameters, i.e. the descriptors contain the properties of the corresponding formal parameters.

- *InDescr* An IN variable parameter.
- *InoutDescr* An IN/OUT variable parameter.

Newqual Denotes the identifier to be used in AS_1. Its name in *Newqual* has changed if the procedure is defined in a service.

40	*ProcessD*	::	*ParameterD* Validinputset Outputset ProcessconnectionD*
41	*Validinputset*	=	*Signalqual*-**set**
42	*Outputset*	=	*Signalqual*-**set**
43	*ParameterD*	=	*Sortqual*
44	*ProcessconnectionD*	=	*Qual* $\overrightarrow{m}$ *Qual*-**set**

ProcessD is a descriptor of a process.

*ParameterD** A list of descriptors used when the process is created, to check for sort compability of the actual parameters, i.e. the descriptors contain the sorts of the corresponding formal parameters.

Signalqual-**set** The sets of valid input signals and valid output signals respectively (the valid output signals are deduced from the process graph).

ProcessconnectionD Is the relation between signal routes and service signal routes for the process. If no service signal routes are specified then the map is empty. It is used for checking of VIA in services.

45 *SortD* :: *Equations*$_1$ [*Parentqual*] [*Expression*$_1$] *Newqual*
46 *Parentqual* = *Sortqual*

SortD is a descriptor of a partial type definition. (newtype).

Equations$_1$ The AS_1 *Equations*$_1$ defined for the sort. These equations are used when another sort inherits from this sort.

Parentqual The parent sort of the sort. If *Parentqual* is **nil**, the sort has no parent. *Parentqual* is used for checking the recursiveness of sort definitions.

Expression$_1$ Is the optional AS_1 expression corresponding to the optional initial variable value which can be specified in the partial sort definition.

Newqual Denotes the identifier to be used in AS_1. Its name in *Newqual* has changed if the partial data type is defined in a service.

47 *SyntypeD* :: *Parentqual Newqual* [*Expression*$_1$] *Range-condition*$_1$

SyntypeD is a descriptor of a syntype.

Parentqual Is the parent sort, i.e. if the parent specified in AS_0 is a syntype, it is the parent of that syntype.

Newqual Denotes the identifier to be used in AS_1. Its name in *Newqual* has changed if the syntype is defined in a service.

Expression$_1$ Is the optional AS_1 expression corresponding to the optional initial variable value which can be specified in the syntype definition

Range-condition$_1$ Is the AS_1 range condition which is used for generating default assignment for syntype variables.

48 *GeneratorD* :: *Genparm*$_0{}^+$ *Properties*$_0$

GeneratorD is a descriptor of a data sort generator.

Genparm$_0{}^+$ The AS_0 list of formal parameters taken from the generator heading.

Properties$_0$ The AS_0 body of the generator.

49 *SynD* :: *Sortqual Expr*$_0$

SynD is a descriptor of a synonym.

Sortqual The sort of the synonym. If the sort in AS_0 is absent, *Sortqual* is derived from the sort of the expression contained in the synonym definition.

Expr$_0$ The AS_0 synonym expression.

50 *VarD* :: *Sortqual* [REVEALED] [EXPORTED] [*Expression*$_1$] *Newqual*

VarD is a descriptor of a variable

Sortqual	The sort of the variable
REVEALED	The optional REVEALED attribute as in AS_0
EXPORTED	The optional EXPORTED attribute as in AS_0
[*Expression*$_1$]	The AS_1 version of the optional initial expression specified when the variable is defined
Newqual	The *Qual* to be used in AS_1. Its name in *Newqual* is different from the original name if the variable is defined in a service.

51 *ImportD* :: *Sortqual*

ImportD is a descriptor of an import variable, containing the sort of the import variable.

52 *ViewD* :: *Qual*

ViewD is a descriptor of an view variable, containing the *Qual* of the corresponding revealed variable.

53 *LiteralD* :: *Result*
54 *Result* = *Sortqual*

LiteralD is a descriptor of an literal containing the sort of the literal.

55 *OperatorD* :: *Sortqual*$^+$ *Result Newqual Explicit*
56 *Explicit* = *Bool*

OperatorD is a descriptor of an operator.

Sortqual$^+$	the list of sort identifiers corresponding to the operator arguments
Result	the sort identifier corresponding to the result.
Newqual	The unique operator identifier used in AS_1
Explicit	This flag is false if the operator is implicitly inherited from another sort.

57 *ValueidD* :: *Mapvalue Sortqual*-set *Explicit*
58 *Mapvalue* = [*Qual*]

ValueidD is a descriptor of an axiom variable (a value identifier).

Mapvalue	If the identifier is introduced by literal quantification, the descriptor contains the current literal value. In the quantified axioms, the value identifier is replaced by this literal value.
Sortqual-set	The set of sort *Qual*s which at any specific time are legal for the value identifier. If the value identifier is introduced by explicit quantification then the set contains only one sort.
Explicit	A flag indicating whether the value identifier is introduced by explicit quantification.

59 *ErrorD* :: ()

ErrorD is used for detecting recursive definitions of synonyms, signal lists and generators. During evaluation of the definitions of these constructs, their descriptors are replaced by a *ErrorD* such that any use inside their own definitions can be detected. It is also used for masking out synonyms which cannot be used in simple expressions of select definitions.

2.2 Description of *Quotdict*

The following entries contain some auxiliary information which only for practical reasons (in order to reduce the number of parameters and results of the Meta IV functions) are included in the *Dict* domain.

1 *Quotdict* = SCOPEUNIT $\overset{m}{\rightarrow}$*Qual* ∪
GLOBALNAMES $\overset{m}{\rightarrow}$*Globalnames* ∪
LABELDICT $\overset{m}{\rightarrow}$*Labeldict* ∪
STATEDICT $\overset{m}{\rightarrow}$*Statedict* ∪
OUTSIGNALS $\overset{m}{\rightarrow}$*Signalqual*-**set** ∪
DATATYPEDEF $\overset{m}{\rightarrow}$*Data-type-definition*$_1$ ∪
SERVICES $\overset{m}{\rightarrow}$(*Statetuplemap Servicetuple*) ∪
IMPLIED $\overset{m}{\rightarrow}$*NameclosureD*-**set** ∪
IMPORTLIST $\overset{m}{\rightarrow}$(*Name*$_0$ *Qual* [*Expr*$_0$])*

2 *Globalnames* = *Emptyqid Formuniquenm Formuniquenm*

3 *Labeldict* = *Label*$_0$ $\overset{m}{\rightarrow}$*Transition*$_0$

4 *Statedict* = *Statename*$_0$ $\overset{m}{\rightarrow}$(*StateD* | *ContenablestateD*)

5 *StateD* :: *Speclist* [*Importstateinf*]

6 *Importstateinf* = [*Statename*$_0$] [*Graph-node*$_1$ | *Decision-node*$_1$]

7 *ContenablestateD* :: *Transition*$_0$

8 *Statetuplemap* = *Statename*$_0$$^+$ $\overset{m}{\rightarrow}$*Statename*$_0$

9 *Servicetuple* = *Servicequal**

10 *Speclist* = *Spec**

11 *Spec* = *Qual Statespec*$_0$

12 *Servicequal* = *Qual*

13 *Emptyqid* = *Qual*

14 *Formuniquenm* = *Name*$_0$

SCOPEUNIT — Contains the *Qual* denoting the current level, i.e. *Qual* denotes the identifier of the enclosing scopeunit.

GLOBALNAMES — Contains a descriptor *Globalnames* containing names of objects which are required for the transformation of enabling condition and continuous signal.

- *emptyqid* — The signal which is send by processes to themselves in continuous signals and enabling conditions. Declared in AS_1 at the system level.
- *formuniquenm* — The names of the two variables used in continuous signals. The first one is updated at each entry to a continuous signal state and its new value is send with the *emptyqid* signal thus forming a new unique value each time. The second one holds the value received by the *emptyqid* signal. Declared in AS_1 in every process.

LABELDICT — *Labeldict* holds information of any labels (connector names) used in a process, procedure or service body. It is constructed before the actual transformation of the body takes place and it contains the *Transition*$_0$s which follows given labels (in connectors).

STATEDICT — *Statedict* contains the states of a body. The transformation of the states is based on *Statedict*. If the state descriptor is a *ContenablestateD* then it denotes a state which contains continuous signal or enabling condition. Such states have no explicit representation in AS_1 (they are represented by a number of synthetic states). The nextstate node is replaced by the transition string leading to the synthetic states.

If the state descriptor is a *StateD* it denotes a state which has an AS_1 representation. It contains:

Speclist — A list of input descriptors for the state. Each input descriptor (*Spec*) contains:

Qual — A *Qualifier* denoting the context in which the subsequent transition should be evaluated. As services are merged into a single *Statedict* before the state bodies are transformed into AS_1, the *Qual* differs for the various *Specs* if the *Statedict* originated from a service decomposition. In other cases *Qual* denotes the identifier of the enclosing process or procedure.

Statespec$_0$ — The AS_0 input node.

Importstateinf — If the state is an implicit state originating from an import expression then *Importstateinf* is present. It contains the name of the previous state (*Statename*$_0$) such that nextstate nodes with a dash following the import expression can be properly substituted. If it is **nil** then the import expression originates from an initial transition. It also contains the AS_1 node wherein the expression containing the import expression occurred. If the expression contained more than one import expression only one of the import state descriptors contains such a node (the others are **nil**).

As opposed to normal states, *Statedict* descriptors for import states are constructed while the states is transformed.

OUTSIGNALS — Contains a set of signal. It is used for deducing the output signals from the process body.

DATATYPEDEF — Contains the AS_1 *Data-type-definition*$_1$ associated to every scope unit. When a partial type definition is encounted, *Data-type-definition*$_1$ is updated with the new sort, operators and equations.

SERVICES — This information is used during the transformation of a *Statedict* formed from a service decomposition. All services are ordered in a tuple *Servicetuple* which in conjunction with *Statetuplemap* is used for associating unique AS_1 state names with the old state names occurring in the various services.

IMPLIED — A set of pairs of variable sort and implicit variable name, collected during the transformation of a process definition. I.e. during transformation of axioms, the implicit quantification names are collected and during the transformation of actions, the implicit import variable names are collected. Note that the *NameclosureD* is not related conceptually to the *NameclosureD* occurring in *Exportmap* although they are the same domain.

IMPORTLIST — During transformation of an expression, information is collected about the import expressions occurring in the expression. In the resulting AS_1 expression, the import expression is replaced by a new implicit name. After the transformation, a number of states (one for each import expression) are generated (and added to *Statedict*). The transition following a new state are generated from the action in which the expression was used, the implicit name (*Name*$_0$), the identifier of the import variable (Qual) and the optional Pid expression used in the import expression (*Expr*$_0$). The length of the tuple denoted by IMPORTLIST equals the number of import expressions occurring in the construct.

2.3 Other Domains

1 *Decl*$_1$ = *Block-definition*$_1$ | *Channel-definition*$_1$ | *Signal-definition*$_1$ | *Signal-route-definition*$_1$ | *Procedure-definition*$_1$ | *Process-definition*$_1$ | *Signal-route-definition*$_1$ | *Syn-type-definition*$_1$ | *Data-type-definition*$_1$ | *Variable-definition*$_1$ | *View-definition*$_1$ | *Timer-definition*$_1$

2 *Context* = CONSTANT | AXIOMS | MAPPING | EXPRESSION

These domains (synonyms) are introduced in order to avoid cumbersome repetitions in the function type specifications:

Decl$_1$ A shorthand for denoting an AS$_1$ definition which many "definition transforming" functions delivers as result.

Context A shorthand for denoting the possible contexts in which expressions are transformed. The context can either be constant (as for instance in the transformation of answers), axioms, the mapping section of the axioms or any other case.

3 *Processqual* = *Qual*

4 *Operatorqual* = *Qual*

These domain names are used in some places for indicating that in the given context the *Qual* (identifier) is of a specific type (i.e. process or operator).

5 *External-Information* = ...

External-Information contains the additional information required to give semantics to SDL. This information is brought in from the outside and it is therefore given as parameter to *definition-of-SDL* (together with the system definition and the predefined sorts). As SDL does not define how the information is structured (say actual generic parameters), the Meta-IV functions which uses *External-Information* are informally defined.

External-Information contains the following.

- Information about actual generic parameters (the corresponding formal parameters are the external synonyms and the informal texts in the option node).
- Information about actual subset parameters to indicate which consistent subset should be selected, i.e. the set of *Block-identifier*$_1$s is deduced from *External-Information* (see *definition-of-SDL* and *select-consistent-subset*).
- Information about the indeterministic delay in channels (i.e. a "random" function).
- Information about the start time and the time unit (used in modelling absolute time).

6 *Auxiliary-Information* = *Time-information Term-Information Is-expiredF DelayF*

7 *Time-information* = (*Ground-term*$_1$ → *Ground-term*$_1$) *Ground-term*$_1$

8 *Term-Information* = *Sort-identifier*$_1$ *Literal-operator-identifier*$_1$ *Literal-operator-identifier*$_1$ *Literal-operator-identifier*$_1$

9 *Is-expiredF* = *Ground-term*$_1$ *Ground-term*$_1$ → *Bool*

10 *DelayF* = () ⇒ *Bool*

Auxiliary-Information contains some information which, apart from a *System-definition*$_1$ and a consistent subset, is required for interpretation. *Auxiliary-Information* is constructed

in the Static Semantics and given as parameter to the *system* processor when it is started (see the function *definition-of-SDL*). *Auxiliary-Information* consist of

Time-information — The function in *Time-information* is used in the Dynamic Semantics for updating the current time. Given an AS_1 literal of the TIME sort it returns another literal of the time sort. In addition, *Time-information* contains the *Ground-term*$_1$ denoting the initial time which is defined outside the SDL system (i.e. it is derived from *External-Information*).

Term-Information — AS_1 contains no information about spelling of identifiers. However, four AS_1 identifiers must be known (i.e. distinguished) in the Dynamic Semantics. These identifiers are PID, NULL, TRUE and FALSE. *Term-information* denotes these identifiers and they are constructed in the Static Semantics.

Is-expiredF — This function is constructed in the Static Semantics and used in the Dynamic Semantics for testing on whether a given timer has expired. Given two AS_1 literals of the time sort, **true** is returned if the value of the first parameter is greater than or equal to the second parameter.

DelayF — This function is used for modelling the indeterministic delay in channels (see the *path* processor in Annex F.3). The function is derived from *External-Information* and it has an imperative nature implying that it may depend on some external physical parameters.

3 Transformation of AS_0 into AS_1

3.1 Main Functions

This section contains the two functions:

definition-of-SDL Which is the outermost function invoked from the outside. When given an SDL system (in AS_0 form) as parameters it defines the semantics (static and dynamic) of the system. It also forms the link between the static- and dynamic semantics.

transform-system Which is the entry function for the static semantics. When given an SDL system (in AS_0 form) as parameter, it returns a specification on the SDL abstract syntax form (AS_1 form) if the AS_0 form is statically correct (well-formed).

definition-of-SDL(*extparms*, *systemdef*, *predefsorts*) $\triangleq$ (3.1.1)

```
1  (let (as1, auxinf) = transform-system(systemdef, predefsorts, extparms) in
2   if as1 = nil then
3     undefined
4    else
5     (let subsetcut = select-consistent-subset(as1, extparms) in
6      start system(as1, subsetcut, auxinf)))
```

type : *External-Information* Sys_0 $Datadef_0{}^+ \Rightarrow$

Objective Define the properties of SDL

Parameters

extparms	Some *External-Information* (see section 2.3).
systemdef	The AS_0-tree representing the SDL system
predefsorts	The predefined data in AS_0 form.

Algorithm

Line 1	Transform the system into the abstract syntax form (AS_1 form).
Line 2	If static errors are found (i.e. if no AS_1 representation could be derived) then the behavior is not defined
Line 4	If no static errors are found then
Line 5	Select the set of *Block-identifier*$_1$s denoting the consistent subset
Line 6	Create a system instance, i.e. create a Meta-IV process which behaves like the underlying system.

transform-system(*genericsystem*, *predefdatasorts*, *extparms*) $\triangleq$ (3.1.2)

```
 1  (let mk-Sys0(sysdef, refdeflist) = apply-generic-parameters(genericsystem, extparms) in
 2   let mk-Sysdef0(snm, decllist, tnm) = sysdef in
 3   let (as0global, globalentities) = as0-global-entities() in
 4   let as1datadef = mk-Data-type-definition1(name-to-name1(create-unique-name()), {}, {}, {}, {}) in
 5   (trap exit with (nil, nil) in
 6    (let d = [⟨(SYSTEM, snm)⟩ ↦ mk-SystemD(),
 7              SCOPEUNIT       ↦ ⟨(SYSTEM, snm)⟩,
 8              GLOBALNAMES     ↦ globalentities,
 9              DATATYPEDEF     ↦ as1datadef] in
10     let decllist' = replace-references(decllist, refdeflist)(⟨(SYSTEM, snm)⟩) in
11     let predefdict be s.t. (, predefdict) = transform-decllist(predefdatasorts)(predefdict + d) in
12     let decllist'' = remove-select(decllist', {})(predefdict) in
13     if (∃fulldict ∈ Dict)
14        ((trap exit with false in
15          (let (dict) = transform-decllist(decllist'' ⌢ ⟨as0global⟩ ⌢ predefdatasorts)(fulldict + d) in
16           fulldict = dict))) then
17       (let (as1dcl, dict) be s.t. (as1dcl, dict) =
18            transform-decllist(decllist'' ⌢ ⟨as0global⟩ ⌢ predefdatasorts)(dict + d) in
19        let as1 = make-as1tree(SYSTEM, snm, as1dcl, nil, nil, nil)(dict) in
20        let auxinf = generate-auxiliary-information(extparms)(dict) in
21        (tnm ∉ {snm, nil}
22           → exit("§2.2.2: Ending name in system definition is different from defining name"),
23         ¬(∃b ∈ elems decllist')(is-Blockdef0(b))
24           → exit("§2.4.2: System definition must contain at least one block"),
25         T → (as1, auxinf)))
26       else
27        (nil, nil))))
```

type: *Sys*$_0$ *Datadef*$_0^+$ *External-Information* → [*System-definition*$_1$] [*Auxiliary-Information*]

Objective Transform an SDL system in AS_0 form into the corresponding AS_1 form.

Parameters

genericsystem	A generic system definition
predefdatasort	A list of predefined data also in AS_0 form
extparameters	The external information (see section 2.3)

Result The AS_1 system definition and some information to be used in Annex F.3. If the system definition is not well-formed, **nil** is returned for both results

Algorithm

Line 1	Transform the generic system definition into a concrete system definition by supplying an expression to every external synonym defined in the system and by transforming every informal text in the option answers into an AS_0 *Conditionlist*$_0$. SDL does not define how to make these transformations.
Line 2	Let *snm* denote the system name, *decllist* denote the definition list and *tnm* denote the tailing name
Line 3	Construct the AS_0 definition and the *Globalnames* descriptor (see *Quotdict*) corresponding to the global emptyq signal.
Line 4	Construct the initial *Data-type-definition*$_1$ for the system scopeunit. When through the definition list (line 17) it contains information about all partial data types defined at the system level.
Line 5	If *transform-decllist* or any of the other functions are trapped, the specification is not well-formed.

Line 6-9 Construct an initial *Dict* consisting of the system descriptor, the SCOPEUNIT entry indicating the system level , the global names and the current *Data-type-definition*$_1$.

Line 10 Insert all the remote definitions (*refdeflist*) in the definition list of the system. The new definition list (*decllist'*) contains no references

Line 11 Transform the predefined data in order to construct the *Dict* consisting of the predefined data sorts only. The result is only used during the removal of SELECT statements (line 12), i.e. the predefined sorts are elsewhere treated as ordinary definitions as shown in line 17. It reads : Let *predefdict*) be such that one of the results (the other result is the AS$_1$ definitions which are not used here) of transforming the predefined sorts in the scope of *predefdict* (i.e. with semantic information of all predefined sorts) is *predefdict* itself. Such a *Dict* do always exist as the predefined sorts are assumed well-formed.

Line 12 Remove all the SELECT statements from the system. Since option nodes does not contain any definitions they can be removed "on the fly" when appropriate.

Line 13-16 If there exist a *Dict* (*fulldict*) such that transformation of *decllist''* in the scope of *fulldict* can be made without causing any errors (i.e. no application of **exit** in *transform-decllist*) and such that the result is *fulldict* itself, then *decllist''* is well-formed.

Line 17 If so, then transform *decllist''* in the scope of such a *Dict*. The other result ($as_1\, dcl$) is the set of AS$_1$ definitions. This way of using *Dict* is essential in the modelling and the following should therefore be noted:

- It is used in order to overcome the problem that names in SDL may be referred before they are defined textually.
- The Meta-IV functions which transform definitions have the full *Dict* available (all descriptors). For instance, *transform-blockdef* returns a *Block-definition*$_1$ and a *Dict* descriptor for the block, even though the block descriptor already is contained in the *Dict* given as parameter to the function. The *Dict* given as parameter is used for accessing properties of entities used by the block, but the block descriptor itself in the *Dict* parameter cannot be used (otherwise it cannot be guaranteed that a *Dict* solution exists for well-formed specifications, which means that the Meta-IV specification might be invalid).
- During the transformation of the definitions the returned descriptors are joined and eventually (when all definitions have been transformed) the result of *transform-decllist* is the same as the parameter.
- *dict* in line 17 is a *Descriptordict* only. There are no contributions from *Quotdict* in the resulting *Dict*.
- Line 11 and line 17 in *transform-system* are the only places in the complete Formal Definition where the **be such that** construct have been used in such a far-reaching way.

Line 19 Construct the *System-definition*$_1$

Line 20 Generate the *Auxiliary-Information* to be used during interpretation

Line 21 If the tailing name is specified then it must be equal to the system name.

Line 23 There must be at least one block definition in the system.

Line 25 Return the AS$_1$ representation of the system and the auxiliary information

as$_0$*-global-entities*() $\triangleq$ **(3.1.3)**

```
1  (let emptyqnm = create-unique-name(),
2       formunique1 = create-unique-name(),
3       formunique2 = create-unique-name() in
4   let emptyqid = mk-Id0(⟨⟩, emptyqnm) in
5   let globalnames = (emptyqid, formunique1, formunique2) in
6   let intgid = mk-Id0(⟨⟩, mk-Name0("INTEGER", nil)) in
7   let as0def = mk-Sigdef0(⟨mk-Sigelem0(emptyqnm, ⟨intgid⟩, nil)⟩) in
8   (as0def, globalnames))
```

type: () → *Sigdef*$_0$ *Globalnames*

Objective Construct a global AS$_0$ definition to be placed on the system level and construct the *Globalnames* closure (see the definition of *Quotdict*) which is of general use during the transformation. The AS$_0$ definition defines the emptyq signal used in enabling condition and continuous signal

Algorithm

Line 1	Create a unique name for the emptyq signal.
Line 2-3	Create unique names for the two variables used in connection with continuous signal. In every process, these two variables are defined.
Line 4	Construct an AS$_0$ identifier for the emptyq signal.
Line 5	Construct the *Globalnames* closure.
Line 6	Construct an AS$_0$ identifier for the integer sort.
Line 7	Construct the AS$_0$ definition for the emptyq signal.
Line 8	Return the AS$_0$ definition and the *Globalnames* closure

3.2 Replacing Definition References

In this section, references are replaced by remote definitions. The entry function *replace-references* takes as argument the definitions from the system level and a list of remote definitions. The result is a definition list containing no references.

replace-references(*deflist*, *remotelist*)(*scopeunit*) $\triangleq$ **(3.2.1)**

```
1  if (∀i1, i2 ∈ ind remotelist)
2     (i1 ≠ i2 ∧
3       cases (remotelist[i1], remotelist[i2]):
4        ((mk-Blockdef0(mk-Id0(q1, nm1),,,), mk-Blockdef0(mk-Id0(q2, nm2),,,)),
5         (mk-Prdef0(mk-Id0(q1, nm1),,,,,,), mk-Prdef0(mk-Id0(q2, nm2),,,,,,)),
6         (mk-Procdef0(mk-Id0(q1, nm1),,,,), mk-Procdef0(mk-Id0(q2, nm2),,,,)),
7         (mk-Servicedef0(mk-Id0(q1, nm1),,,,), mk-Servicedef0(mk-Id0(q2, nm2),,,,)),
8         (mk-Chansubdef0(mk-Id0(q1, nm1),,), mk-Chansubdef0(mk-Id0(q2, nm2),,))
9           → nm1 = nm2 ⊃ (q1 ≠ q2 ∧ q1 ≠ ⟨⟩ ∧ q2 ≠ ⟨⟩),
10        T → true)) then
11    (let (deflist', remoteset) = remove-references(deflist, elems remotelist, false)(scopeunit) in
12     if remoteset = {} then
13       deflist'
14      else
15       exit("§2.4.1: Remote definition is not referenced in the system definition"))
16   else
17    exit("§2.4.1: Remote definitions are not unique")
```

type: *Decl*$_0^+$ *Decl*$_0^*$ → *Qual* → *Decl*$_0^+$

Objective Replace all references in the system by remote definitions

Parameters

deflist	The list of definitions for the system level.
remotelist	The list of remote definitions.

Result The definition list of the system level containing no references.

Algorithm

Line 1-10	For every two different remote block definitions, process definitions, procedure definitions, service definitions or substructure definitions it must hold that if they have the same name in their identifier then their qualifiers must be distinct and non-empty.
Line 11-15	Replace the references by their definitions. The resulting list is *deflist'*. No remote definitions may be left when all references have been replaced.

remove-references(*dlist*, *remoteset*, *chansub*)(*unit*) $\triangleq$ (3.2.2)

```
1  if dlist = ⟨⟩ then
2    (⟨⟩, remoteset)
3   else
4    (let match-id(nm, id) = s-Name0(id) = nm ∧ s-Qualifier0(id) ∈ {⟨⟩, unit} in
5     cases hd dlist:
6     (mk-Blockref0(name)
7        → if (∃blkdef ∈ remoteset)(is-Blockdef0(blkdef) ∧ match-id(name, s-Blockid0(blkdef))) then
8            (let blkdef ∈ remoteset be s.t. is-Blockdef0(blkdef) ∧ match-id(name, s-Blockid0(blkdef)) in
9             let mk-Blockdef0(bid, decll, sub, tid) = blkdef in
10            if tid ∉ {nil, bid} then
11              exit("§2.4.1: Ending and starting identifier in the definition are different")
12            else
13             (let blkdef' =
14                  mk-Blockdef0(mk-Id0(⟨⟩, name), decll, sub, mk-Id0(⟨⟩, name)) in
15              remove-references(⟨blkdef'⟩ ⌢ tl dlist, remoteset \ {blkdef}, chansub)(unit)))
16          else
17           exit("§2.4.1: No remote definition matches reference"),
18     mk-Prref0(name, inst)
19       → if (∃prdef ∈ remoteset)(is-Prdef0(prdef) ∧ match-id(name, s-Prid0(prdef))) then
20           (let prdef ∈ remoteset be s.t. is-Prdef0(prdef) ∧ match-id(name, s-Prid0(prdef)) in
21            let mk-Prdef0(pid, inst', b, d, e, f, tid) = prdef in
22            let inst'' = select-remote-number-of-instances(inst, inst') in
23            if tid ∉ {nil, pid} then
24              exit("§2.4.1: Ending and starting identifier in the definition are different")
25            else
26             (let prdef' =
27                  mk-Prdef0(mk-Id0(⟨⟩, name), inst'', b, d, e, f, mk-Id0(⟨⟩, name)) in
28              remove-references(⟨prdef'⟩ ⌢ tl dlist, remoteset \ {prdef}, chansub)(unit)))
29          else
30           exit("§2.4.1: No remote definition matches reference"),
31     mk-Procref0(name)
32       → if (∃procdef ∈ remoteset)(is-Procdef0(procdef) ∧ match-id(name, s-Procid0(procdef))) then
33           (let procdef ∈ remoteset be s.t. is-Procdef0(procdef) ∧ match-id(name, s-Procid0(procdef)) in
34            let mk-Procdef0(pid, parm, decll, body, tid) = procdef in
35            if tid ∉ {nil, pid} then
36              exit("§2.4.1: Ending and starting identifier in the definition are different")
37            else
38             (let procdef' =
39                  mk-Procdef0(mk-Id0(⟨⟩, name), parm, decll, body, mk-Id0(⟨⟩, name)) in
40              remove-references(⟨procdef'⟩ ⌢ tl dlist, remoteset \ {procdef}, chansub)(unit)))
41          else
42           exit("§2.4.1: No remote definition matches reference"),
43     mk-Serviceref0(name)
44       → if (∃servicedef ∈ remoteset)(is-Servicedef0(servicedef) ∧
45                                      match-id(name, s-Serviceid0(servicedef))) then
46           (let servicedef ∈ remoteset be s.t. is-Servicedef0(servicedef) ∧
47                match-id(name, s-Serviceid0(servicedef)) in
48            let mk-Servicedef0(sid, a, b, d, tid) = servicedef in
49            if tid ∉ {nil, sid} then
50              exit("§2.4.1: Ending and starting identifier in the definition are different")
51            else
52             (let servicedef' =
53                  mk-Servicedef0(mk-Id0(⟨⟩, name), a, b, d, mk-Id0(⟨⟩, name)) in
54              remove-references(⟨servicedef'⟩ ⌢ tl dlist, remoteset \ {servicedef}, chansub)(unit)))
55          else
56           exit("§2.4.1: No remote definition matches reference"),
```

```
57      mk-Chansubref_0(name)
58        → if (∃chansubdef ∈ remoteset)(is-Chansubdef_0(chansubdef) ∧
59                                       s-Chansubid_0(chansubdef) ≠ nil ∧
60                                       match-id(name, s-Chansubid_0(chansubdef))) then
61            (let chansubdef ∈ remoteset be s.t. is-Chansubdef_0(chansubdef) ∧
62                 match-id(name, s-Chansubid_0(chansubdef)) in
63             let mk-Chansubdef_0(cid, decll, tid) = chansubdef in
64             if tid ∉ {nil, cid} then
65               exit("§2.4.1: Ending and starting identifier in the definition are different")
66             else
67              (let chansubdef' =
68                   mk-Chansubdef_0(mk-Id_0(⟨⟩, name), decll, mk-Id_0(⟨⟩, name)) in
69               remove-references(⟨chansubdef'⟩ ⌢ tl dlist, remoteset \ {chansubdef}, true)(unit)))
70          else
71           exit("§2.4.1: No remote definition matches reference"),
72      mk-Blocksubref_0(name)
73        → if (∃chansubdef ∈ remoteset)(is-Chansubdef_0(chansubdef) ∧
74                                       s-Chansubid_0(chansubdef) ≠ nil ∧
75                                       match-id(name, s-Chansubid_0(chansubdef))) then
76            (let chansubdef ∈ remoteset be s.t. is-Chansubdef_0(chansubdef) ∧
77                 match-id(name, s-Chansubid_0(chansubdef)) in
78             let mk-Chansubdef_0(bid, decll, tid) = chansubdef in
79             if tid ∉ {nil, bid} then
80               exit("§2.4.1: Ending and starting identifier in the definition are different")
81             else
82              (let blksubdef' =
83                   mk-Blocksubdef_0(mk-Id_0(⟨⟩, name), decll, mk-Id_0(⟨⟩, name)) in
84               remove-references(⟨blksubdef'⟩ ⌢ tl dlist, remoteset \ {chansubdef}, chansub)(unit)))
85          else
86           exit("§2.4.1: No remote definition matches reference"),
87      mk-Decomposition_0(list)
88        → (let (list', rrest) = remove-references(list, remoteset, false)(unit) in
89           (mk-Decomposition_0(list'), rrest)),
90      mk-Blockdef_0(mk-Id_0(q, nm), decll, sub, tid)
91        → (let (decll', rrest) = remove-references(decll, remoteset, false)(unit ⌢ ⟨(BLOCK, nm)⟩) in
92           let (sub', rrest') =
93               if sub = nil
94                 then (nil, rrest)
95                 else remove-references(⟨sub⟩, rrest, false)(unit ⌢ ⟨(BLOCK, nm)⟩) in
96           let (drest', rrest'') = remove-references(tl dlist, rrest', chansub)(unit) in
97           (⟨mk-Blockdef_0(mk-Id_0(q, nm), decll', sub', tid)⟩ ⌢ drest', rrest'')),
98      mk-Blocksubdef_0(id, decll, tid)
99        → (let (, scnm) = unit[len unit] in
100          let mk-Id_0(q, nm) = if id = nil then mk-Id_0(⟨⟩, scnm) else id in
101          let unit' = unit ⌢ ⟨(SUBSTRUCTURE, nm)⟩ in
102          let (decll', rrest) = remove-references(decll, remoteset, false)(unit') in
103          (mk-Blocksubdef_0(mk-Id_0(q, nm), decll', tid), rrest)),
104     mk-Chandef_0(nm, a, b, sub, tnm)
105       → (let (sub', rrest) =
106              if sub = nil
107                then (nil, remoteset)
108                else remove-references(⟨sub⟩, remoteset, chansub)(unit) in
109          let (drest', rrest') = remove-references(tl dlist, rrest, chansub)(unit) in
110          (⟨mk-Chandef_0(nm, a, b, sub', tnm)⟩ ⌢ drest', rrest')),
```

```
111     mk-Chansubdef0(id, decll, tid)
112       → (let (, scnm) = unit[len unit] in
113          let mk-Id0(q, nm) = if id = nil then mk-Id0(⟨⟩, scnm) else id in
114          let unit' = unit ⌢ ⟨(SUBSTRUCTURE, nm)⟩ in
115          let (decll', rrest) = remove-references(decll, remoteset, true)(unit') in
116          (mk-Chansubdef0(mk-Id0(q, nm), decll', tid), rrest)),
117     mk-Prdef0(mk-Id0(q, nm), a, b, d, decll, body, tid)
118       → (let (decll', rrest) =
119              remove-references(decll, remoteset, false)(unit ⌢ ⟨(PROCESS, nm)⟩) in
120          let (body', rrest') =
121             if is-Body0(body) then
122               (body, rrest)
123              else
124               remove-references(⟨body⟩, rrest, false)(unit ⌢ ⟨(PROCESS, nm)⟩) in
125          let (drest', rrest'') = remove-references(tl dlist, rrest', chansub)(unit) in
126          (⟨mk-Prdef0(mk-Id0(q, nm), a, b, d, decll', body', tid)⟩ ⌢ drest', rrest'')),
127     mk-Procdef0(mk-Id0(q, nm), p, decll, body, tid)
128       → (let (decll', rrest) =
129              remove-references(decll, remoteset, false)(unit ⌢ ⟨(PROCEDURE, nm)⟩) in
130          let (drest', rrest') = remove-references(tl dlist, rrest, chansub)(unit) in
131          (⟨mk-Procdef0(mk-Id0(q, nm), p, decll', body, tid)⟩ ⌢ drest', rrest')),
132     mk-Servicedef0(mk-Id0(q, nm), a, decll, body, tid)
133       → (let (decll', rrest) =
134              remove-references(decll, remoteset, false)(unit ⌢ ⟨(SERVICE, nm)⟩) in
135          let (drest', rrest') = remove-references(tl dlist, rrest, chansub)(unit) in
136          (⟨mk-Servicedef0(mk-Id0(q, nm), a, decll', body, tid)⟩ ⌢ drest', rrest')),
137     mk-Select0(expr, decll)
138       → if is-wf-entities(decll, chansub)(unit) then
139            (let (decll', rrest) = remove-references(decll, remoteset, chansub)(unit) in
140             let (drest', rrest') = remove-references(tl dlist, rrest, chansub)(unit) in
141             (⟨mk-Select0(expr, decll')⟩ ⌢ drest', rrest'))
142           else
143            exit("§4.3.3: The definitions in select are not allowed in that scopeunit"),
144     T → (let (drest, rrest) = remove-references(tl dlist, remoteset, chansub)(unit) in
145          (⟨hd dlist⟩ ⌢ drest, rrest))))
```

type: $(Decl_0^* \mid Blocksubdef_0 \mid Chansubdef_0 \mid Decomposition_0)\ Refdecl_0^*\ Bool \rightarrow Qual \rightarrow (Decl_0^* \mid Blocksubdef_0 \mid Chansubdef_0 \mid Decomposition_0)\ Refdecl_0^*$

Objective Replace every reference in a definition list by a remote definition.

Parameters

dlist	The AS_0 definition list
remoteset	The set of remote definitions.
chansub	Is **true** only if *dlist* denotes a definition list for a channel substructure. This parameter is used for checking of correct $Connect_0$s (see *is-wf-entities*).
unit	The context (represented by a qualifier) in which the definition list occurs.

Result The AS_0 definition list containing no references, and the set of remote definitions for which no reference has been found. removed.

Algorithm

Line 1	When the definition set (*dlist*) is empty, return no definition and the modified set of remote definitions. (*remove-references* recursively returns the complete definition list)
Line 4	Define a utility function which returns **true** if a *nm* ($Name_0$) in a reference matches an *id* (from a definition), i.e. if the name in *id*

	equals the $Name_0$ in the reference and the qualifier either is empty or equal to the scopeunit where the reference is placed.
Line 5	Consider the first definition in the definition list.
Line 6	If the definition is a block reference then there must exist a remote block definition which have the same name in its identifier as the block reference (checked by *match-id*, see line 4) and for which the qualifier in its identifier either is empty or equal to the scopeunit where the reference is placed.
Line 8	Let *blkdef* denote that block definition.
Line 10-11	The ending identifier in the remote block definition must either be omitted or equal to the starting identifier.
Line 13	Reconstruct the block definition with empty qualifiers in the starting and ending identifiers
Line 15	Add it to the list of definitions such that enclosed definitions in the block are dealt with in the next recursion level and remove the block definition from the set of remote definitions.
Line 18-30	Same scheme as for referencing block definitions as shown above except for line 22 where the $Instances_0$ for the reference is compared with $Instances_0$ for the remote definition
Line 31-42	Same scheme as for referencing block definitions as shown above.
Line 43-56	Same scheme as for referencing block definitions as shown above.
Line 57-71	Same scheme as for referencing block definitions as shown above. In line 59, it is checked that a channel substructure identifier is present if the channel substructure definition is remote.
Line 72-86	Same scheme as for referencing block definitions as shown above, except that block sub-structure definitions syntactically appears as the more general channel sub-structure definitions so the block sub-structure is converted.
Line 87-89	If the definition is a decomposition then replace references in the definition list contained in the decomposition.
Line 90	If the definition is a block definition then
Line 91	Remove references in the definition list contained in the block.
Line 92	Remove references in the block sub-structure.
Line 96	Remove references in the rest of the definition list which contained the block definition and
Line 97	Return the modified block definition concatenated with the other modified definitions and return the modified remote definition list.
Line 98-103	Remove references in a block sub-structure definition.
Line 104-110	Remove references in a channel definition.
Line 111-116	Remove references in a channel sub-structure definition.
Line 117-126	Remove references in a process definition. If the process contains a decomposition (tested on line 120) then also references in the decomposition must be replaced.
Line 127-131	Remove references in a procedure definition
Line 132-136	Remove references in a service definition.
Line 137-141	Remove references contained in a select definition and check that every definition in the select is allowed in the enclosing scopeunit.
Line 144	In other cases, the definition is returned unchanged.

select-remote-number-of-instances(*inst*, *reminst*) $\triangleq$ (3.2.3)

```
1  (inst = nil ∧ reminst ≠ nil
2     → reminst,
3   reminst = inst ∨ reminst = nil
4     → inst,
5   T → exit("§2.4.4: Remote number of instances specification does not match process reference"))
```

type: [*Instances*$_0$] [*Instances*$_0$] → [*Instances*$_0$]

Objective Select the *Instances*$_0$ for a process in the case where the process is referenced.

Parameters

inst	The *Instances*$_0$ specified for the process reference.
reminst	The *Instances*$_0$ specified for the remote definition.

Algorithm

Line 1	If *Instances*$_0$ is specified for the remote definition only then return that *Instances*$_0$ specification
Line 3	If *Instances*$_0$ for the process reference equals *Instances*$_0$ for the remote definition or *Instances*$_0$ is specified for the reference only then return that *Instances*$_0$ specification.
Line 5	Otherwise the *Instances*$_0$ specification for the process is inconsistent.

is-wf-entities(*decllist*, *ischannelsub*)(*level*) $\triangleq$ (3.2.4)

```
1  if decllist = ⟨⟩ then
2    true
3   else
4    (let (q,) = level[len level] in
5     is-wf-entities(tl decllist, ischannelsub)(level) ∧
6      cases hd decllist:
7       (mk-Blockdef0(,,,),
8        mk-Chandef0(,,,,)
9          → q ∈ {SYSTEM, SUBSTRUCTURE},
10       mk-Connect0(id,)
11         → q ∈ {SUBSTRUCTURE, BLOCK} ∧ (id = ENV ⊃ (q = SUBSTRUCTURE ∧ ischannelsub)),
12       mk-Sigroutedef0(,),
13       mk-Prdef0(,,,,,,)
14         → q = BLOCK,
15       mk-Signallistdef0(,),
16       mk-Sigdef0()
17         → q ∉ {SERVICE, PROCEDURE},
18       mk-Procdef0(,,,,),
19       mk-Vardef0(,,)
20         → q ∈ {SERVICE, PROCESS, PROCEDURE},
21       mk-Importdef0(),
22       mk-Viewdef0(),
23       mk-Timerdef0()
24         → q ∈ {SERVICE, PROCESS},
25       T → true))
```

type: *Decl*$_0^*$ *Bool* → *Qual* → *Bool*

Objective Check that a definition list originating from a select definition is syntactically allowed in a given context. However, decomposition is not treated here.

Parameters

decllist	The definition list to be checked.
ischannelsub	A flag which is **true** if *decllist* belongs to a channel sub-structure.

Result True if allowed

Algorithm

Line 1	When through then return **true**.
Line 4	Extract the scopeunit type (*q*) from the *Qual* denoting the current scopeunit.
Line 5-25	The definition list is well-formed if the first definition is well-formed (line 7-25) and the rest of the definition list is well-formed (line 5).
Line 7-8	If the first definition is a block definition or a channel definition then the scopeunit of the context must be the system or a block sub-structure or a channel sub-structure.
Line 10	If it is a connection then the scopeunit must either be a block sub-structure a channel substructure or a block.
Line 12-13	If it is a signal route definition of a process definition then the scopeunit must be a block.
Line 15-16	If it is a signal list definition or a signal definition then the scopeunit must not be a service or a procedure.
Line 18-19	If it is a procedure definition or a variable definition then the scopeunit must be either a service or a process or a procedure.
Line 21-23	If it is an import definition or a view definition or a timer definition then the scopeunit must be a service or a process

3.3 Removal of Select Definitions

In this section, select definitions are evaluated. The entry function is *remove-select* applied in *transform-system*. For each definition list the following steps are taken:

1. Collect the names of the sorts which are defined in the definition list, including the sorts defined in a contained select definition. Add the sorts to the sorts collected in the surrounding scopeunit. This is necessary in order to trap the mess if predefined sorts are redefined.

2. Collect all synonyms defined in the definition list or in definition lists of contained select definitions. Put an *ErrorD* descriptor in *Dict* for these synonyms indicating that they cannot be used yet. Those of these synonyms which are defined by using any of the collected (visible) sorts or which have more than one definition (they are also multiple defined if they are defined in different select definitions) are identified. They cannot be used in select definitions at all since using them could result in different results depending of the order of selection. Note that it is only during removal of select definitions that these synonyms cannot be used. Afterward, they are not necessarily multiple defined.

3. Remove the select definitions. Each time a select definition has been replaced by its contained definition list, the descriptors for the synonyms in the definition list are changed from an *ErrorD* to a *SynD*, except for those which are not allowed to be used.

4. When all select definitions have been removed, *SynD* descriptors are made for all synonyms in the resulting definition list (also for the synonyms which previously could not be used) and the sorts defined in the resulting definition list are collected.

5. Select definitions in enclosed scopeunits are removed using the updated dict and using the collected sorts.

As transition options do not contain any definitions, they can be evaluated while the state-body is transformed into AS_1.

remove-select(*decllist*, *sursorts*)(*dict*) $\triangleq$ (3.3.1)

```
1  (let sorts = sursorts ∪ collect-sorts(decllist) in
2   let (allsyn, illsyn) = collect-illegal-synonyms(decllist, sorts) in
3   let dict' = [dict(SCOPEUNIT) ⌢ ⟨(VALUE, nm)⟩ ↦ mk-ErrorD() | nm ∈ allsyn] in
4   let decllist' = remove-select-in-decllist(decllist, illsyn)(dict') in
5   let dict'' = repeat-collecting-synonyms(decllist', {})(dict) in
6   let sorts' = sursorts ∪ collect-sorts(decllist') in
7   ⟨remove-select-in-enclosed-scopeunit(decllist[i], sorts')(dict'') | 1 ≤ i ≤ len decllist⟩)
```

type: $(Decl_0 \mid Blocksubdef_0 \mid Decomposition_0)^*$ $Name_0$-**set** $\rightarrow$ *Dict* $\rightarrow$
$(Decl_0 \mid Blocksubdef_0 \mid Decomposition_0)^*$

Objective Expand the select definitions for a scopeunit and for all select definitions in scopeunits contained in the definition list.

Parameters

decllist The definition list containing select definitions. Block sub-structure and service decomposition are for convenience also considered as definitions in this context.

sursorts The set of sorts (excluding the predefined) which potentially are visible in the definition list (i.e. possibly depending on whether they are selected by a contained select definition). This information is required in order to achieve correct treatment in the case where the predefined data are redefined.

dict The *Dict*, which only contains descriptors for the predefined sorts and for the visible, but non-local, synonyms of the predefined sorts.

Result The definition list where all select definitions have been expanded.

Algorithm

Line 1 Collect the sorts which are defined in the definition list or are defined in a select definition.

Line 2 Let *allsyn* denote the synonyms which are defined in the definition list or are defined in a contained select definition. Let *illsyn* (which is a subset of *allsyn*) denote the synonyms which cannot be used in removing of select definitions for this definition list.

Line 3 Put an *ErrorD* descriptor in *Dict* to indicate that the locally defined synonyms cannot be used yet.

Line 4 Remove the select definitions.

Line 5 Put a *SynD* descriptor in *Dict* for the synonyms in the resulting definition list.

Line 6 Collect the sorts again, this time using the resulting definition list.

Line 7 Remove select definitions in contained scopeunits.

remove-select-in-decllist(*decllist*, *illsyn*)(*dict*) $\triangleq$ (3.3.2)

```
1  (let dict' = repeat-collecting-synonyms(decllist, illsyn)(dict) in
2   if (∃d ∈ elems decllist)(is-Select0(d)) then
3     (if (∃d ∈ elems decllist)(is-Select0(d) ∧ is-wf-simple-expr(s-Expr0(d))(dict')) then
4        (let d ∈ elems decllist be s.t. is-Select0(d) ∧ is-wf-simple-expr(s-Expr0(d))(dict') in
5         let mk-Select0(expr, decllist') = d in
6         let selected = eval-simple-expr(expr, "BOOLEAN")(dict') in
7         let decllist'' = ⟨d' ∈ elems decllist | d' ≠ d⟩ in
8         if selected then
9            remove-select-in-decllist(decllist' ⌢ decllist'', illsyn)(dict)
10          else
11           remove-select-in-decllist(decllist'', illsyn)(dict))
12       else
13        exit("§4.3.2: Simple expression is not of a predefined sort or contains undefined identifiers"))
14    else
15     decllist)
```

type : $(Decl_0 \mid Blocksubdef_0 \mid Decomposition_0)^*$ $Name_0$-**set** $\rightarrow$ *Dict* $\rightarrow$
$(Decl_0 \mid Blocksubdef_0 \mid Decomposition_0)^*$

Objective Remove the select definitions for a definition list.

Parameters

decllist	The definition list containing select definitions
illsyn	The synonyms which cannot be used in simple expressions of select definitions.
dict	The *Dict*, which only contains descriptors for the predefined sorts and for the visible synonyms (*SynD* or *ErrorD*).

Result The definition list where all select definitions have been removed.

Algorithm

Line 1	Update *dict* with descriptors for the synonyms (of the predefined sorts) which are defined in the definition list (but not contained in a select definition)
Line 2	If there (still) exist a select definition in the definition list then
Line 3	If the expression in the select definition is well-formed then
Line 4	Let *d* denote that select definition
Line 6	Evaluate the simple expression in *d*.
Line 7	Construct a definition list where *d* has been removed.
Line 8	If the definitions in the select definition is selected then
Line 9	Continue removing select definitions where the definition list is extended to include the definitions from the select definition.
Line 11	If the select definition is not selected, then continue with the definition list where the select definition has been removed
Line 15	If all select definitions in the definition list have been removed then return the definition list.

eval-simple-expr(*expr*, *sort*)(*dict*) $\triangleq$ (3.3.3)

```
1  (let predefqual = get-predef-sort(sort)(dict) in
2   let (as1 tree, , ) = transform-expr(expr, CONSTANT, {predefqual})(dict) in
3    eval-expr(as1 tree, sort)(dict))
```

type : $Expr_0$ $Char^+$ $\rightarrow$ *Dict* $\rightarrow$ (*Intg* | *Bool*)

Objective Evaluate a simple expression

Parameters

expr	The AS_0 simple expression
sort	The spelling of the sort of the expression. The sort is either "Boolean" or "Integer".

Result The Meta-IV types *Intg* if the SDL sort is "Integer" and the Meta-IV domain *Bool* if the SDL sort is "Boolean".

Algorithm

Line 1	Construct the *Qual* of the sort
Line 2	Transform the expression into AS_1 notifying that the expression is (must be) a constant (ground) expression and that the expression is (must be) of the specified sort.
Line 3	Evaluate the AS_1 expression (this function is not formally defined)

repeat-collecting-synonyms(*decllist*, *illsyn*)(*dict*) $\triangleq$ **(3.3.4)**

```
1  (let dict' = collect-synonyms(decllist, illsyn)(dict) in
2   if dict' = dict then dict' else repeat-collecting-synonyms(decllist, illsyn)(dict'))
```

type: $Decl_0^*$ $Name_0$-**set** $\rightarrow$ *Dict* $\rightarrow$ *Dict*

Objective Collect descriptors for all the well-formed synonyms (i.e. for the synonyms which at this stage do not depend on a synonym defined in a select definition).

Parameters

decllist	The definition list to be inspected
illsyn	The synonyms from *decllist* which cannot be used, i.e. their *ErrorD* descriptor should not be changed into a *SynD* descriptor.

Result The *Dict* updated with descriptors for the well-formed synonyms

Algorithm

Line 1	Collect descriptors for the well-formed synonyms in the definition list *decllist* by going through the list.
Line 2	If no more synonym can be changed in *dict'* then return *dict* else
Line 2	Go through the list one more time. As synonyms may be mutually dependent more synonym definitions may now be well-formed (in the context of *dict'*).

collect-synonyms(*decllist*, *illsyn*)(*dict*) $\triangleq$ (3.3.5)

```
1  if decllist = ⟨⟩ then
2    dict
3   else
4    (if is-Synonymdef0(hd decllist) then
5       (let mk-Synonymdef0(nm, sort, expr) = hd decllist in
6        let qualset = {qual ∈ dom dict | is-SortD(dict(qual))} in
7        let sortset =
8            if sort = nil then
9              qualset
10           else
11            (trap exit with {} in
12             {get-parent(get-visible-qual(sort, TYPE)(dict))(dict)}) in
13       let synqual = dict(SCOPEUNIT) ⌢ ⟨(VALUE, nm)⟩ in
14       let (as1tree, qset, ) =
15           if sortset = {} then
16             (nil, {}, )
17           else
18            (trap exit with (nil, {}, ) in
19             transform-expr(expr, CONSTANT, sortset)(dict + [synqual ↦ mk-ErrorD()])) in
20       if as1tree = nil ∨ card qset ≠ 1 ∨ nm ∈ illsyn then
21         collect-synonyms(tl decllist, illsyn)(dict)
22        else
23         (let sortqual ∈ qset in
24          let d = [synqual ↦ mk-SynD(sortqual, expr)] in
25          collect-synonyms(tl decllist, illsyn)(dict + d)))
26     else
27      collect-synonyms(tl decllist, illsyn)(dict))
```

type : $Decl_0^*$ $Name_0$-**set** $\rightarrow$ *Dict* $\rightarrow$ *Dict*

Objective Go through a definition list and collect descriptors for all the well-formed synonyms

Parameters

decllist	The definition list containing synonym definitions
illsyn	The synonyms from *decllist* which cannot be used, i.e. their *ErrorD* descriptor should not be changed into a *SynD* descriptor.
dict	The *Dict* which contains descriptors for the predefined sorts and for the visible synonyms.

Result A *Dict* which is updated with *SynD* descriptors for the well-formed synonyms.

Algorithm

Line 1	When through then return the (updated) *Dict*.
Line 4	If the next definition in the list is a synonym definition then
Line 6	Extract the *Quals* (identifiers) of the predefined sorts from *dict*.
Line 7-12	If a sort is specified in the synonym definition then the set of legal sorts for the expression contains only that sort (if the specified sort is not well-formed then the set is empty), otherwise the set of legal sorts include all the predefined sorts.
Line 13	Let *synqual* denote the *Qual* of the synonym.
Line 14-19	Try to transform the expression into AS_1. If it fails it is either a static error (will be caught later) or the expression contains synonyms which have not been considered yet (they might be defined in a select definition or defined in **tl** *decllist*).

Line 20 — If it fails i.e. if the AS_1 expression *as1 tree* is not present or the sort of the expression is indeterminable (the cardinality of the returned set of sorts is different from 1) or if the synonym is one of those which cannot be used then continue with the rest of the definition list else

Line 23 — Let the sort of the expression be denoted by *sortqual.*

Line 24-25 — Update *dict* with the synonym descriptor and go through the rest of the definition list.

Line 27 — If the first definition in the definition list is not a synonym definition then continue with the rest of the list.

•

is-wf-simple-expr(*expr*)(*dict*) $\triangleq$ (3.3.6)

```
1  (let boolqual = get-predef-sort("BOOLEAN")(dict) in
2   trap exit with false in
3   (let (,,) =transform-expr(expr, CONSTANT, {boolqual})(dict) in
4    true))
```

type: $Expr_0 \rightarrow Dict \rightarrow Bool$

Objective — Check whether a simple boolean expression is well-formed. If it is not well-formed, it may be because it is defined in terms of synonyms which have not been incorporated in *dict* yet.

Parameters

expr — The AS_0 expression to be checked

Result — True if well-formed

Algorithm

Line 1 — Extract the *Qual* (identifier) of the boolean sort. The set of legal sorts for the expression only includes that sort.

Line 2-3 — If the transformation function is trapped, then false is returned otherwise

Line 4 — True is returned

remove-select-in-enclosed-scopeunit(*decl*, *sorts*)(*dict*) $\triangleq$ (3.3.7)

```
(let level = dict(SCOPEUNIT) in
 cases decl:
  (mk-Blockdef0(mk-Id0(q, nm), decll, blksub, tid)
     → (let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(BLOCK, nm)⟩] in
        (let (decll', blksub') =
            if blksub = nil then
              (remove-select(decll, sorts)(dict'), nil)
             else
              (let dlist = remove-select(decll ⌢ ⟨blksub⟩, sorts)(dict') in
               let i be s.t. is-Blocksubdef0(dlist[i]) in
               (⟨dlist[n] | 1 ≤ n ≤ len dlist ∧ n ≠ i⟩, dlist[i])) in
         mk-Blockdef0(mk-Id0(q, nm), decll', blksub', tid))),
  mk-Servicedef0(mk-Id0(q, nm), sigl, decll, body, tid)
     → (let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(SERVICE, nm)⟩] in
        let decll' = remove-select(decll, sorts)(dict') in
        mk-Servicedef0(mk-Id0(q, nm), sigl, decll', body, tid)),
  mk-Chandef0(nm, p1, p2, chansub, tnm)
     → if chansub = nil then
         decl
        else
         (let mk-Chansubdef0(id, decll, tid) = chansub in
          let nm' = if id = nil then nm else s-Name0(id) in
          let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(SUBSTRUCTURE, nm')⟩] in
          let decll' = remove-select(decll, sorts)(dict') in
          mk-Chandef0(nm, p1, p2, mk-Chansubdef0(id, decll', tid), tnm)),
  mk-Prdef0(mk-Id0(q, nm), inst, parm, inpset, decll, body, tid)
     → (let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(PROCESS, nm)⟩] in
        let (decll', body') =
            if is-Body0(body) then
              (remove-select(decll, sorts)(dict'), body)
             else
              (let dlist = remove-select(decll ⌢ ⟨body⟩, sorts)(dict') in
               let i be s.t. is-Decomposition0(dlist[i]) in
               (⟨dlist[n] | 1 ≤ n ≤ len dlist ∧ n ≠ i⟩, dlist[i])) in
        mk-Prdef0(mk-Id0(q, nm), inst, parm, inpset, decll', body', tid)),
  mk-Blocksubdef0(id, decll, tid)
     → (let (, nm) = level[len level] in
        let nm' = if id = nil then nm else s-Name0(id) in
        let dict' = dict + [SCOPEUNIT ↦ level ⌢ ⟨(SUBSTRUCTURE, nm')⟩] in
        let decll' = remove-select(decll, sorts)(dict') in
        mk-Blocksubdef0(id, decll', tid)),
  mk-Decomposition0(decll)
     → (let decll' = remove-select(decll, sorts)(dict) in
        if (∀d ∈ elems decll')(is-Sigroutedef0(d) ∨
                               (is-Connect0(d) ∧ is-Id0(s-Connectpoint0(d))) ∨
                               is-Servicedef0(d)) then
          mk-Decomposition0(decll')
         else
          exit("§4.3.3: The selected definition is not allowed in that scopeunit")),
  T → decl))
```

type : (*Decl*$_0$ | *Blocksubdef*$_0$ | *Decomposition*$_0$) *Name*$_0$-**set** → *Dict* → (*Decl*$_0$ | *Blocksubdef*$_0$ | *Decomposition*$_0$)

Objective Remove select definitions in an enclosed scopeunit.

Parameters

decl A definition which may be a scopeunit.

sorts The sorts visible (excluding the predefined).

Result A definition wherein select definitions have been removed.

Algorithm

Line 1 Denote the context (in which the definition occurs) by *level.*

Line 3-12 If the definition is a block definition then remove the select definitions in the definition list of the block (line 7). If the block contains a sub-structure then it is also considered as a definition. The select definitions are removed in the context of the block, i.e. SCOPEUNIT is *Dict* is updated to denote the block.

Line 13-41 Do the same for the other alternative scopeunits as done in the case where the definition is a block definition.

Line 42-49 If the definition is a decomposition (which may contain select definitions although it is not a scopeunit) then check that the resulting definition list (after removal of select definitions), only contains service signal route definitions, service signal route connections and service definition. The $connectpoint_0$ in a service signal route connection must not be ENV (this case is not checked syntactically if the connection is contained in a select definition).

collect-illegal-synonyms(*decllist*, *sorts*) $\triangleq$ (3.3.8)

```
1  (if decllist = ⟨⟩ then
2     ({}, {})
3   else
4    (let (restsyn, restillsyn) = collect-illegal-synonyms(tl decllist, sorts) in
5     let (synset, illsynset) =
6          cases hd decllist:
7          (mk-Synonymdef0(nm, sort, )
8             → if (sort ≠ nil ∧ s-Name0(sort) ∈ sorts ∧ s-Qualifier0(sort) ≠ ⟨(SYSTEM, )⟩) then
9                  ({nm}, {nm})
10                else
11                 ({nm}, {}),
12           mk-Select0(, dlist)
13             → collect-illegal-synonyms(dlist, sorts),
14           T → ({}, {})) in
15     (restsyn ∪ synset, restillsyn ∪ illsynset ∪ (restsyn ∩ synset))))
```

type: $Decl_0^*$ $Name_0$-set → $Name_0$-set $Name_0$-set

Objective Collect the synonyms which (potentially) are defined in a definition list and identify those of these synonyms which cannot be used in any simple expressions of select definitions.

Parameters

decllist The definition list

sorts The sorts which (potentially) are defined in the definition list.

Result The set of potentially defined synonyms and the synonyms which cannot be used.

Algorithm

Line 1 When through the definition list, return empty sets.

Line 4 Collect the synonym sets for the rest of the definition list.

Line 5 Let *synset* denote the synonyms defined in this definition and let *illsynset* denote those of these which cannot be used.

Line 7-11 If the definition is a synonym definition then the synonym name is *synset* and the name cannot be used if the sort name is one of the visible sorts and the qualifier does not denote the system level. Note that the *Dict* do only contain the descriptors for the predefined sorts. This check is therefore only of importance if a predefined sort has been redefined.

Line 12 If the definition is a select definition then collect the synonym sets for the contained definitions.

Line 15 The resulting set of defined synonym names is the names for the rest of definitions joined with the names for this definition. The resulting set of non-usable synonyms is the non-usable synonyms for the rest of definitions joined with the names for this definition joined with the names both defined in the rest of definition and in this definition.

collect-sorts(*decllist*) $\triangleq$ (3.3.9)

```
1  (if decllist = ⟨⟩ then
2     {}
3   else
4    (collect-sorts(tl decllist) ∪
5      cases hd decllist:
6      (mk-Syntypedef0(nm,,,,),
7       mk-Partialtypedef0(nm,,,,)
8          → {nm},
9       mk-Select0(, dlist)
10         → collect-sorts(dlist),
11      T → {})))
```

type: $Decl_0^* \rightarrow Name_0$-**set**

Objective Collect the sort names for the sorts which potentially are defined in a definition list

Parameters

decllist The definition list

Result The set of potentially defined sort names (whether they actually are defined depends on select definitions).

Algorithm

Line 1 When through the definition list, return the empty set.

Line 6-7 If the definition is a partial type definition or a syntype definition then extract the sort name.

Line 9 If the definition is a select definition then collect the potentially defined sort names in the contained definition list.

3.4 Transformation of Definitions

transform-decllist(*decllist*)(*dict*) $\triangleq$ (3.4.1)

```
1  (if decllist = ⟨⟩ then
2     ({}, [])
3   else
4    (let (as1 dcl, d) = transform-decl(hd decllist)(dict) in
5     let (as1 dcll, d') = transform-decllist(tl decllist)(dict) in
6     if dom d ∩ dom d' ≠ {} then
7       exit("§2.2.2: Two definitions in the same scopeunit and same entity class define the same name")
8      else
9       (as1 dcl ∪ as1 dcll, d + d')))
```

type: $Decl_0^* \rightarrow Dict \rightarrow Decl_1$-set $Dict$

Objective Transform a list of AS_0 definitions into a set of AS_1 definitions.

Parameters An AS_0 definition list.

Result The AS_1 definitions and the *Dict* contributions from the AS_0 definition list. Note that it is not the entire *dict* which is returned, as opposed to the expression and graph transforming functions. This means that no returned descriptor (except for *ValueidD* which is used strictly local in axioms and generator parameters and *ErrorD* which is used locally to trap recursive definitions) influence the content of *dict*. The equivalence between the returned descriptors and *dict* is fulfilled by the **be such that** construct in the function *transform-system* as mentioned earlier.

Algorithm

Line 4 Transform the first definition in the definition list.

Line 5 Transform the rest of the definitions.

Line 6 The pair of name and entity class (reflected in the *Qual*) of the first definition must be disjoint from the pairs representing the rest of the definitions

Line 9 Return the *Dict* contributions and the AS_1 definitions.

transform-decl(*decl*)(*dict*) $\triangleq$ (3.4.2)

```
 1  ( cases decl:
 2    (mk-Blockdef0(,,,)
 3       → transform-blockdef(decl)(dict),
 4     mk-Chandef0(,,,,)
 5       → transform-channeldef(decl)(dict),
 6     mk-Prdef0(,,,,,,)
 7       → transform-processdef(decl)(dict),
 8     mk-Sigdef0()
 9       → transform-signaldef(decl)(dict),
10     mk-Procdef0(,,,,)
11       → transform-proceduredef(decl)(dict),
12     mk-Partialtypedef0(,,,,)
13       → transform-partial-typedef(decl)(dict),
14     mk-Syntypedef0(,,,,)
15       → transform-syntype(decl)(dict),
16     mk-Sortgenerator0(,,,,)
17       → transform-sortgenerator(decl)(dict),
18     mk-Synonymdef0(,,)
19       → transform-synonymdef(decl)(dict),
20     mk-Vardef0(,,)
21       → transform-vardef(decl)(dict),
22     mk-Viewdef0()
23       → transform-viewdef(decl)(dict),
24     mk-Importdef0()
25       → transform-importdef(decl)(dict),
26     mk-Sigroutedef0(,,)
27       → if is-BlockD(dict(dict(LEVEL)))
28            then transform-signalroutedef(decl)(dict)
29            else transform-servicesigroutedef(decl)(dict),
30     mk-Signallistdef0(,)
31       → transform-signallistdef(decl)(dict),
32     mk-Timerdef0()
33       → transform-timerdef(decl)(dict),
34     mk-Servicedef0(,,,,)
35       → build-service-descriptor(decl)(dict),
36     T → ({}, [])))
```

type: $Decl_0 \rightarrow Dict \rightarrow Decl_1$-**set** *Dict*

Objective Transform an AS_0 definition into an AS_1 definition.

Result See *transform-decllist*.

Algorithm Transform either

Line 2 A block definition or
Line 4 A channel definition or
Line 6 A process definition or
Line 8 A signal definition or
Line 10 A procedure definition or
Line 12 A partial data type definition or
Line 14 A syn type definition or
Line 16 A sort generator or
Line 18 A synonym definition or
Line 20 A variable definition or
Line 22 A view definition or
Line 24 An import definition or

Line 26-29	A signal route definition or a service signal route depending on the enclosing scopeunit or
Line 30	A signal list definition or
Line 32	A timer definition or
Line 29	A service signal route definition or
Line 34	A service definition
Line 19	Nothing, as the other kind of definitions (i.e. connects) are handled elsewhere.

3.4.1 Block Definitions

transform-blockdef(mk-$Blockdef_0$(*bid*, *decllist*, *subdef*, *tid*))(*dict*) $\triangleq$ (3.4.1.1)

```
 1  (let mk-Id0(q, bnm) = bid in
 2   let bqual = dict(SCOPEUNIT) ⌢ ⟨(BLOCK, bnm)⟩ in
 3   let dict' = dict + [SCOPEUNIT ↦ bqual] +
 4       [DATATYPEDEF ↦ initialdatadef(dict)] in
 5   let explicit = (∃d ∈ elems decllist)(is-Sigroutedef0(d)) in
 6   let (chandefl, routedefl, connects, exp, imp) =
 7       implicit-channels-and-signal-routes(decllist)(dict') in
 8   let (outerchannels, cdi) = transform-decllist(chandefl)(dict) in
 9   let (as1 dcl, di) = transform-decllist(decllist ⌢ routedefl)(dict') in
10   let (as1 connect, connectmap) = transform-block-connect(decllist, {}, [])(dict') in
11   if ((∃i ∈ elems decllist)(is-Prdef0(i)) ∨ subdef ≠ nil) ∧ q = ⟨⟩ ∧ tid ∈ {bid, nil} then
12     if subdef = nil then
13       (let descr = [bqual ↦ mk-BlockD(exp, imp, explicit, connectmap)] in
14        let as1 block = make-as1 tree(BLOCK, bnm, as1 dcl ⌢ as1 connect, nil, nil, nil)(dict) in
15        (outerchannels ∪ {as1 block}, descr + di + cdi))
16      else
17       (let mk-Blocksubdef0(subid, subdecll, tailid) = subdef in
18        let mk-Id0(q', bnm') = if subid = nil then mk-Id0(⟨⟩, bnm) else subid in
19        let chansubqual = dict(SCOPEUNIT) ⌢ ⟨(SUBSTRUCTURE, bnm')⟩ in
20        let subqual = if chansubqual ∈ dom dict then
21                        chansubqual
22                      else
23                        bqual ⌢ ⟨(SUBSTRUCTURE, bnm')⟩ in
24        let dict'' = [SCOPEUNIT     ↦ subqual,
25                      DATATYPEDEF ↦ initialdatadef(dict')] in
26        let (as1 subdcl, subdi) = transform-decllist(subdecll)(dict'') in
27        let as1 connect' = transform-substructure-connect(subdecll ⌢ connects, {}, [])(dict'') in
28        let as1 tree = make-as1 tree(SUBSTRUCTURE, bnm',
29                                     as1 subdcl ∪ as1 connect', nil, nil, nil)(subdi) in
30        let as1 block' = make-as1 tree(BLOCK, bnm, as1 dcl, as1 tree, nil, nil)(dict) in
31        let descr = [bqual ↦ mk-BlockD(exp, imp, explicit, connectmap)] +
32            (if chansubqual ∈ dom dict then [] else [subqual ↦ mk-BlocksubD()]) in
33        (q' ≠ ⟨⟩
34           → exit("§2.4.1: Defining names may only be qualified in remote definitions"),
35         tailid ∉ {nil, subid}
36           → exit("§2.2.2: Ending name in block sub-structure definition is different from defining name"),
37         ¬(∃d ∈ elems subdecll)(is-Blockdef0(d))
38           → exit("§3.2.2: Block sub-structure does not contain a block definition"),
39         T → (outerchannels ∪ {as1 block'}, descr + di + subdi + cdi)))
40    else
41     (¬(∃i ∈ elems decllist)(is-Prdef0(i) ∨ subdef ≠ nil)
42        → exit("§2.4.3: Block must contain either one or more processes or a sub-structure definition"),
43      q ≠ ⟨⟩
44        → exit("§2.4.1: Defining names may only be qualified in remote definitions"),
45      T → exit("§2.2.2: Ending name in block definition is different from defining name")))
```

type : $Blockdef_0 \rightarrow Dict \rightarrow Decl_1$-set $Dict$

Objective Transform an AS_0 block definition into an AS_1 block definition.

Parameters The AS_0 block definition containing

bid	The unqualified block identifier
decllist	Its definition list
subdef	Its optional sub-structure
tid	Its optional ending identifier

Result See *transform-decllist.*

Algorithm

Line 2-3	Construct the *Qual* denoting the block identifier (*bqual*) and update the *Dict* entry SCOPEUNIT to denote the context of the block. Construct the initial *Data-type-definition*$_1$ for the block.
Line 5	Let *explicit* be **true** if explicit signal routes are specified for the block.
Line 6	Create the implicit AS_0 channel definitions implied by export and import from and to the block, and also the *ExpimpchanD* maps (see the domain definition of *ExpimpchanD*) which is used to interface the channel definitions to the sub-structure and to the surroundings.
Line 8	Transform the implicit channel definitions into AS_1.
Line 9	Transform the contained definition list, with the *dict* entry SCOPEUNIT updated to denote the block.
Line 10	Transform the channel to signal route connections in the block. Also return the *BlockconnectionD* for the block.
Line 11	If the sub-structure is omitted then there must exist at least one process definition in the block. The block identifier must not be qualified and if the ending identifier is specified then it must be equal to the block identifier.
Line 12-15	If the block does not contain a block sub-structure then return the AS_1 definition of the implicit channels (*outerchannels*) and of the block (as_1 *block*) and *Dict* contributions for the implicit channels (*cdi*), the block (*descr*), and the contained definitions. (*di*)
Line 17	Decompose the block sub-structure.
line 18	If the block sub-structure name is omitted then it is the same as the block name. Let *bnm'* denote the block sub-structure name
Line 19-20	Let *subqual* denote the *Qual* of the sub-structure. If the block sub-structure is derived from a channel sub-structure then *subqual* denotes the *Qual* of the channel sub-structure.
Line 24	Update SCOPEUNIT and DATATYPEDEF as in line 3.
Line 26	Transform the definition list of the sub-structure.
Line 27	Create the AS_1 definition of the block (as_1 *block*),
Line 28	Create the AS_1 connections from the AS_0 connections in *subdecll.*
Line 30	The AS_1 definition of the block sub-structure (as_1 *tree*),
Line 31-32	The *Dict* entry for the block (*bqual*) and unless the sub-structure denotes a channel sub-structure then also the *Dict* entry for the block sub-structure (*subqual*).
Line 33	The block sub-structure name (identifier) must not be qualified (as it is not a remote definition).
Line 35	If the tailing id is specified, it must be the same as the sub-structure identifier (*subid*).
Line 37	The sub-structure must contain at least one block definition.
Line 39	Return the AS_1 definitions as in line 15 and *Dict* contributions for the implicit channels (*cdi*), for the block and the sub-structure (*descr*), and the contained definitions in the block (*di*) and the definitions contained in the sub-structure (*subdi*).
Line 41	There must exist at least one process definition in the block unless a sub-structure is specified and
Line 33	The block identifier must not be qualified and
Line 45	If the ending name is specified then it must be equal to the block name

initialdatadef(*dict*) $\triangleq$ (3.4.1.2)

```
1  (let mk-Data-type-definition1(unm, , , , ) = dict(DATATYPEDEF),
2       qual = dict(SCOPEUNIT) ↷ ⟨(TYPE, unm)⟩ in
3   let typeid = make-as1-identifier(qual)(dict) in
4   mk-Data-type-definition1(create-unique-name(), {typeid}, {}, {}, {}))
```

type : *Dict* → *Data-type-definition*$_1$

Objective

Construct the AS_1 data type definition for a scopeunit where the identifier, representing the data type definition of the surrounding scopeunit, is included in the type union. Literals, operators and equations are added to the data type definition during the transformation of the partial data type definitions in the scopeunit

Algorithm

Line 1 Extract the AS_1 data type definition of the surrounding scopeunit.

Line 2 Construct the *Qual* of the type identifier of the surrounding scopeunit.

Line 3 Construct the AS_1 identifier of the type.

Line 4 Return a new data type definition which has a unique name and where the type identifier of the surrounding scopeunit is included in the type union and which contains no literals, operators or equations

3.4.2 Channel Definitions

transform-channeldef(mk-$Chandef_0$(*cnm*, *chanpath*, *ochanpath*, *csub*, *tnm*))(*dict*) $\triangleq$ (3.4.2.1)

```
 1  (let mk-Chanpath0(ori, dest, siglist) = chanpath in
 2   let orid = get-visible-qual(ori, BLOCK)(dict),
 3       destid = get-visible-qual(dest, BLOCK)(dict),
 4       sigidset = transform-signallist(siglist)(dict),
 5       sigidset' =
 6       if ochanpath = nil then
 7         {}
 8        else
 9         (let mk-Chanpath0(orig', dest', osiglist) = ochanpath in
10          let orid' = get-visible-qual(orig', BLOCK)(dict),
11              destid' = get-visible-qual(dest', BLOCK)(dict) in
12          if orid = destid' ∧ destid = orid' then
13            transform-signallist(osiglist)(dict)
14           else
15            exit("§2.5: Second path in channel definition must denote reverse direction of first path")) in
16   if orid = destid then
17     exit("§2.5: Endpoints of channel must be different")
18    else
19     (let as1orid = if ori = ENV then ENVIRONMENT else make-as1-identifier(orid)(dict),
20          as1destid = if dest = ENV then
21                        ENVIRONMENT
22                      else
23                        make-as1-identifier(destid)(dict),
24          as1sigs = make-as1idset(sigidset)(dict),
25          as1sigs' = make-as1idset(sigidset')(dict),
26          cqual = dict(SCOPEUNIT) ↷ ⟨(CHANNEL, cnm)⟩ in
27      (¬is-local(orid)(dict) ∨ ¬is-local(destid)(dict)
28         → exit("§2.5: Endpoint of channel is defined in another scopeunit than the channel"),
29       tnm ∉ {cnm, nil}
30         → exit("§2.2.2: Ending name in channel definition is different from defining name"),
31       csub ≠ nil
32         → transform-channel-sub(csub, cnm, sigidset, sigidset', orid, destid)(dict),
33       T → (let descr = [cqual ↦ mk-ChannelD(orid, destid, sigidset, sigidset', nil)] in
34            let path1 = mk-Channel-path1(as1orid, as1destid, as1sigs),
35                path2 = if as1sigs' = {} then
36                          nil
37                        else
38                          mk-Channel-path1(as1destid, as1orid, as1sigs') in
39            ({mk-Channel-definition1(name-to-name1(cnm), path1, path2)}, descr)))))
```

type: $Chandef_0 \rightarrow Dict \rightarrow Decl_1$-set $Dict$

Objective Transform a channel definition into AS_1

Result The AS_1 channel definition and a *dict* contribution containing the channel descriptor and a channel sub-structure descriptor (if present)

Parameters The AS_0 channel definition containing

cnm	The channel name
chanpath	The first path
ochanpath	The second optional channel path
csub	The optional channel sub-structure
tnm	The name ending the definition

Algorithm

Line 1 Decompose the first path

line 2-15	Make the *Qual* of the origination block (*orid*), the destination block (*destid*), the signals (*sigidset*) and the signals in the reverse direction (*sigidset'*).
line 9-15	If the channel is bidirectional then the originating endpoint of one of the directions must be the same as the terminating endpoint of the other direction.
line 16	The originating block must not be the same as the destination block.
line 19-25	Make the $Identifier_1$s of the origination block (as_1 *orid*), the destination block (as_1 *destid*), the signals (as_1 *sigidset*) and the signals in the reverse direction (as_1 *sigidset'*).
line 26	Update SCOPEUNIT to denote the channel identifier.
line 27-28	The originating block and the destination block must be defined at the same level as where the channel is defined and
line 29	The name ending the channel definition must be equal to the channel name if specified
line 31	If a channel sub-structure is present then transform the channel sub-structure else
Line 33	Construct the *Dict* contribution for the channel.
Line 34-35	Construct the two AS_1 paths.
Line 39	Return the AS_1 channel definition and its *Dict* contribution.

transform-channel-sub(*csub*, *nm*, *sigset*, *osigset*, *endpoint*1, *endpoint*2)(*dict*) $\triangleq$ (3.4.2.2)

```
1  (let mk-Chansubdef0(subid, decllist, tailid) = csub in
2   let mk-Id0(q, nm') = if subid = nil then mk-Id0(⟨⟩, nm) else subid in
3   let mk-Id0(qtail, nmtail) = if tailid = nil then mk-Id0(q, nm') else tailid in
4   if q ≠ ⟨⟩ ∨ qtail ≠ ⟨⟩ then
5     exit("§2.4.1: Defining names may only be qualified in remote definitions")
6    else
7     if nmtail ≠ nm' then
8       exit("§2.2.2: Ending name in channel sub-structure definition is different from defining name")
9      else
10      (let level = dict(SCOPEUNIT) in
11       let newbnm = create-unique-name() in
12       let newbqual = level ⌢ ⟨(BLOCK, newbnm)⟩ in
13       let newc1 = create-unique-name(),
14           newc2 = create-unique-name() in
15       let path1 = mk-Chanpath0(as0-id(endpoint1), as0-id(newbqual), ⟨as0-id(q) | q ∈ sigset⟩),
16           opath1 =
17           if osigset = {}
18             then nil
19             else mk-Chanpath0(as0-id(newbqual), as0-id(endpoint1), ⟨as0-id(q) | q ∈ osigset⟩),
20           path2 = mk-Chanpath0(as0-id(newbqual), as0-id(endpoint2), ⟨as0-id(q) | q ∈ sigset⟩),
21           opath2 =
22           if osigset = {}
23             then nil
24             else mk-Chanpath0(as0-id(endpoint2), as0-id(newbqual), ⟨as0-id(q) | q ∈ osigset⟩) in
25       let chandef1 = mk-Chandef0(newc1, path1, opath1, nil, newc1),
26           chandef2 = mk-Chandef0(newc2, path2, opath2, nil, newc2) in
27       let substructurequal = level ⌢ ⟨(SUBSTRUCTURE, nm')⟩ in
28       let dict' = dict + [SCOPEUNIT ↦ substructurequal] in
29       let decllist' = replace-connects(decllist, newc1, endpoint1, newc2, endpoint2)(dict') in
30       let newchannels = (endpoint1, newc1, endpoint2, newc2) in
31       let blockdef =
32           mk-Blockdef0(mk-Id0(⟨⟩, newbnm), ⟨⟩, mk-Blocksubdef0(mk-Id0(⟨⟩, nm'), decllist', nil), nil) in
33       let channelqual = level ⌢ ⟨(CHANNEL, nm)⟩ in
34       let di = [channelqual      ↦ mk-ChannelD(endpoint1, endpoint2, sigset, osigset, newchannels),
```

```
35          substructureequal ↦ mk-ChannelsubD(newbqual)] in
36      let (as1 declset, dict') = transform-decllist((chandef1, chandef2, blockdef))(dict) in
37      (as1 declset, dict' + di)))
```

type : *Chansubdef*$_0$ *Name*$_0$ *Signalqual*-**set** *Signalqual*-**set** *Endpoint Endpoint* → *Dict* → *Decl*$_1$-**set** *Dict*

Objective Transform a channel sub-structure definition into two channel definitions and a block definition

Parameters

csub	The AS_0 channel sub-structure definition
nm	The name of the enclosing channel definition
siglist	The signals conveyed by the enclosing channel.
osiglist	The optional signals leading in the opposite direction.
endpoint1	The block from which *siglist* is sent.
endpoint2	The block to which *siglist* is sent. The *osiglist* signals are sent from *endpoint2* to *endpoint1*.

Result See *transform-decllist*

Algorithm

Line 1	Let *subid* denote the channel sub-structure identifier, *decllist* the definition list enclosed in the sub-structure and *tailid* the identifier ending the sub-structure definition
Line 2	If no identifier (name) is specified for the sub-structure, the name of the surrounding channel is inherited.
Line 3	Let *qtail* denote the qualifier in the tailing identifier (*subid*) and let *nmtail* denote the name in the tailing identifier
Line 4	The name of the sub-structure must not be qualified (as it is not a remote definition) and the identifier ending the definition must be equal to the identifier starting the definition if specified
Line 7	The tailing name must be equal to the channel substructure name
Line 10	Let *level* denote the *Qual* of the block where the channel is defined
Line 11	Create a distinct new name for the block to be constructed
Line 12	Construct the *Dict* entry for the block to be constructed
Line 13-14	Create distinct new names for the two channels to be constructed
Line 15-16	Construct the paths for the first of the two new channels
Line 20-21	Construct the paths for the second of the two new channels
Line 25-26	Construct the two new AS_0 channels
Line 29	Replace the channel endpoint connections (*Connect*$_0$s) by block sub-structure connections (*Connect*$_0$s) in the definition list of the channel sub-structure
Line 31	Construct the new block. It has an empty definition list and a block sub-structure containing the modified definition list
Line 34	Let *di* denote the *Dict* consisting of the descriptor for the enclosing channel and the descriptor for the channel sub-structure.
Line 36	Transform the two constructed channel definitions and the block definition into AS_1
Line 37	Return the three AS_1 definitions and the *Dict* which includes the descriptors for the three definitions, descriptors for the contained definitions, descriptor for the enclosing channel and descriptor for the channel sub-structure.

replace-connects(*decllist*, *newc1*, *endpoint1*, *newc2*, *endpoint2*)(*dict*) $\triangleq$ **(3.4.2.3)**

```
1  (let conset = {d ∈ elems decllist | is-Connect0(d)} in
2   if card conset = 2 then
3     (let {mk-Connect0(blkid1, clist1), mk-Connect0(blkid2, clist2)} = conset in
4      let bqual1 = get-visible-qual(blkid1, BLOCK)(dict),
5          bqual2 = get-visible-qual(blkid2, BLOCK)(dict) in
6      if {bqual1, bqual2} = {endpoint1, endpoint2} then
7        (let (c1, c2) = if bqual1 = endpoint1 then (newc1, newc2) else (newc2, newc1) in
8         let decllist' = ⟨decllist[i] | i ∈ ind decllist ∧ ¬is-Connect0(decllist[i])⟩ in
9         decllist' ⌢
10        ⟨mk-Connect0(mk-Id0(⟨⟩, c1), clist1),
11         mk-Connect0(mk-Id0(⟨⟩, c2), clist2)⟩)
12       else
13        exit("§3.2.3: The block identifiers in the connects do not denote channel endpoints"))
14    else
15     exit("§3.2.3: There must be exactly two channel endpoint connects"))
```

type : $Decl_0^*$ $Channame_0$ *Endpoint* $Channame_0$ *Endpoint* $\rightarrow$ *Dict* $\rightarrow$ $Decl_0^+$

Objective Change a channel sub-structure definition list to a block sub-structure definition list such that the list contains channel connections instead of channel endpoint connections.

Parameters

decllist	The definition list to be changed
newc1,newc2	The names of the two synthetic channels which replaces the channel containing a channel sub-structure.
endpoint1, endpoint2	The endpoints of the channel containing a channel sub-structure. *newc1* is connected to *endpoint1* and *newc2* is connected to *newc2*.

Result The new definition list to be enclosed by a synthetic block sub-structure.

Algorithm

Line 1	Extract the channel endpoint connections from the definition list.
Line 2 and 13	There must be exactly two such channel endpoint connections.
Line 3	Decompose the set containing the two channel endpoint connections
Line 4-5	Construct the *Quals* of the block identifiers mentioned in the two channel endpoint connections
Line 6	These two block identifiers must be the same block identifiers as mentioned in the channel definition.
Line 7	Let *c1* denote the name of the channel to which *clist1* correspond and let *c2* denote the name of the channel to which *clist2* correspond
Line 8	Construct the definition list where the two channel endpoint connections have been removed and
Line 9-11	Return this list concatenated with two synthetic channel connections

3.4.3 Process Definitions

transform-processdef(mk-*Prdef*$_0$(*pid*, *ins*, *pl*, *sigl*, *decll*, *body*, *tid*))(*dict*) $\triangleq$ (3.4.3.1)

```
 1  (let mk-Id0(q, name) = pid in
 2   let mk-Instances0(ini, maxi) = if ins = nil then mk-Instances0(nil, nil) else ins in
 3   let ini' = if ini = nil then 1 else eval-simple-expr(ini, "INTEGER")(dict),
 4       max' = if maxi = nil then infinite else eval-simple-expr(maxi, "INTEGER")(dict) in
 5   (q ≠ ⟨⟩
 6      → exit("§2.4.1: Defining names may only be qualified in remote definitions"),
 7    tid ∉ {pid, nil}
 8      → exit("§2.2.2: Ending name in process definition is different from defining name"),
 9    ini' > max'
10      → exit("§2.2.3: Initial number is greater than maximum number of instances"),
11    ini' < 0
12      → exit("§2.2.3: Initial number of instances is less than zero"),
13    max' = 0
14      → exit("§2.2.3: Maximum number of instances equals zero"),
15    T → (let d = [IMPLIED      ↦ {},
16                  OUTSIGNALS   ↦ {},
17                  SCOPEUNIT    ↦ dict(SCOPEUNIT) ⌢ ⟨(PROCESS, name)⟩,
18                  DATATYPEDEF ↦ initialdatadef(dict)] in
19         let (as1list, pd, dict') = transform-processparm(pl)(dict + d),
20             sigset = transform-validinputset(sigl)(dict + d),
21             (as1declset, ddict) = transform-decllist(decll)(dict + d) in
22         if dom dict' ∩ dom ddict ≠ {} then
23           exit("§2.2.2: Names of formal parameters must be distinct from variable names")
24          else
25           (let (servicedeclset1, as1body, bdict, connectmap) =
26                transform-process-body(body)(dict + d),
27                outsigs = bdict(OUTSIGNALS),
28                number = mk-Number-of-instances1(ini', max') in
29            if (∃q1, q2 ∈ outsigs ∪ sigset)(len q1 > len q2 ∧ ⟨q1[i] | 1 ≤ i ≤ len q2⟩ = q2) then
30              exit("§3.3: Process uses signals on different refinement levels of the same signal")
31             else
32              (let (as1declset', vdict) = make-implicit-vardecl(bdict),
33                   as1totset = as1declset ∪ as1declset' ∪ servicedeclset1,
34                   as1tree =
35                   make-as1tree(PROCESS, name, as1totset, number, as1list, as1body)(ddict) in
36               let delem = [d(SCOPEUNIT) ↦ mk-ProcessD(pd, sigset, outsigs, connectmap)] +
37                   dict' + ddict + vdict in
38               ({as1tree}, delem))))))
```

type: *Prdef*$_0$ → *Dict* → *Process-definition*$_1$-**set** *Dict*

Objective Transform a process definition into AS_1

Parameters The AS_0 process definition containing

pid	The process identifier
ins	The "number of instances" construct
pl	The formal parameter list
sigl	The valid input signal set
decll	The contained definition list
body	The process state body
tid	The tailing identifier

Result See *transform-decllist*

Algorithm

Line 1 Let *name* denote the process name

Line 2	If the "number of instances" construct is omitted it is the same as omitting the initial number of instances and omitting the maximum number of instances
Line 3-4	If the initial number of instances is unspecified, it is equal to 1. If the maximum number of instances is unspecified, an infinite number of instances is allowed. The predefined constant *infinite* denotes an "unlimited" number (see also annex F.1 section 5.8). If one of the expressions are specified, it is evaluated and it must be of the predefined data sort INTEGER, i.e. the integer *Qual* (*iqual*) is given as parameter to *eval-simple-expr* as the only valid sort in the set of sorts.
Line 5	The name (identifier) of the process must not be qualified (as it is not a remote definition)
Line 7	The identifier ending the process definition (*tid*), must be equal to the process name (identifier) if specified
Line 9	The maximum number of instances must be greater than or equal to the initial number of instances
Line 11	The initial number of instances must not be less than zero
Line 13	The maximum number of instances must not be zero
Line 15-16	Initiate the IMPLIED entry to contain no implicit variables and initiate OUTSIGNALS to contain no output signals. The two sets are filled during the transformation of the process body.
Line 17-18	Update SCOPEUNIT to indicate the process level and construct the initial *Data-type-definition*$_1$ for the process.
Line 19	Transform the process formal parameters.
Line 20	Extract the complete valid input signal set.
Line 21	Transform the contained definitions
Line 22	The names of the formal parameters must be different from the variable and synonym names defined in the process (i.e. they must have distinct *Quals*)
Line 25	Transform the process body and return the list of AS_1 definitions from the contained services (if any), the AS_1 process graph, a *Dict* wherein only the IMPLIED entry and the OUTSIGNALS entry is used and the relation between signal routes and service signal routes (*connectmap* is empty if no services are present in the process).
Line 29-30	There must not exist two output signals where the first one is defined in the second one. The same must hold for input signals
Line 32	Construct the implicit AS_1 variable definitions.
Line 33-38	Return the AS_1 process definition (*as*$_1$ *tree*), and the *Dict* contribution consisting of the process descriptor (*delem*), the formal parameter descriptors (*dict'*), and the descriptors of the entities defined inside the process (*ddict*).

transform-validinputset(*sigset*)(*dict*) $\triangleq$ (3.4.3.2)

```
 1  (let qual = dict(SCOPEUNIT) in
 2   let mk-BlockD(, , explicit, ) = dict(get-sur(process-level(qual))) in
 3   let routesigset =
 4       if ¬explicit then
 5         {}
 6        else
 7         {squal | (∃mk-SignalrouteD(p1, p2, s1, s2) ∈ rng dict)
 8                  ((p1 = qual ∧ squal ∈ s2) ∨ (p2 = qual ∧ squal ∈ s1))} in
 9   let squalset =
10       if sigset = nil then
11         {}
12        else
13         (let mk-Inputset0(sigl) = sigset in
14          if sigl = nil then {} else transform-signallist(sigl)(dict)) in
15   let implicit =
16       if is-ServiceD(dict(qual)) then
17         {}
18        else
19         (let (expimpinput, ) = import-export-signals(dict) in
20          expimpinput) in
21   let timers = {tqual ∈ dom dict | is-TimerD(dict(tqual)) ∧ is-local(tqual)(dict)} in
22   let locals = {tqual ∈ dom dict | is-SignalD(dict(tqual)) ∧ is-local(tqual)(dict)} in
23   if squalset ∩ timers = {} ∧ locals ⊆ squalset then
24     routesigset ∪ squalset ∪ implicit ∪ timers
25    else
26     exit("§2.5.2: Valid input signal must contain all local signals and no timers"))
```

type: $[Signallist_0] \rightarrow Dict \rightarrow Signalqual\text{-}set$

Objective Extract the complete valid input signal set for a process or for a service

Parameters

sigset The optional list of AS_0 signal identifiers

Algorithm

Line 1 Let *qual* denote the *Qual* of the service or process

Line 2 *explicit* is **true** if explicit signal routes are specified for the surrounding block.

Line 3-8 Let *routeset* denote the signals contained in a signal route (implicit or explicit) leading to the process or service. The set is empty if no explicit signal routes are specified for the surrounding block.

Line 9-14 Let *squalset* denote the signals contained in the valid input signal set construct for the process or service

Line 15-19 Unless the scopeunit is a service extract the *Quals* of the implicit signals leading to the scopeunit.

Line 21 Let *timers* denote the timer signals defined in the process or service

Line 22 Let *locals* denote the signals defined in the process or service

Line 23 The valid input signal set for the process or service must not include any timers and it must contain all locally defined signals.

Line 24 The complete valid input signal set consist of the signals in the signal routes, the signals in the valid input signal set construct, the implicit signals and the timers.

transform-processparm(*pl*)(*dict*) $\triangleq$ (3.4.3.3)

```
1  (if pl = ⟨⟩ then
2     (⟨⟩, ⟨⟩, [])
3    else
4     (let mk-Parm0(nml, tid) = hd pl in
5      let (as1 list, tp', d') = transform-processparm(tl pl)(dict) in
6      let as0 vardef = mk-Vardefelem0(nml, tid, nil) in
7      let (,d) = transform-vardef(mk-Vardef0(nil, nil, ⟨as0 vardef⟩))(dict) in
8      let tq = get-visible-qual(tid, TYPE)(dict) in
9      let tq' = get-parent(tq)(dict) in
10     let tql = ⟨tq' | 1 ≤ i ≤ len nml⟩ in
11     let as1 id = make-as1-identifier(tq')(dict) in
12     let as1 nml = ⟨name-to-name1(nml[i]) | 1 ≤ i ≤ len nml⟩ in
13     if card elems as1 nml ≠ len as1 nml then
14       exit("§2.2.2: Two definitions in the same scopeunit use the same name")
15      else
16       (let as1 tree = mk-Process-formal-parameter1(as1 nml, as1 id) in
17        (⟨as1 tree⟩ ⌢ as1 list, tql ⌢ tp', d + d'))))
```

type : $Parm_0^* \rightarrow Dict \rightarrow Process\text{-}formal\text{-}parameter_1^*\ Sortqual^*\ Dict$

Objective Transform a list of formal process parameters into AS_1

Parameters

pl — The list of formal AS_0 process parameters.

Result The AS_1 formal parameters, the corresponding sort list which is used for checking of actual parameters in the create request node, and the *Dict* contribution containing the descriptors of the formal parameters.

Algorithm The function traverse recursively through the list of formal parameters.

Line 1 — When through, return empty lists and the empty map

Line 4 — Let *nml* denote the name list of the first element in the list and let *tid* denote their sort

Line 5 — Transform the rest of the formal parameters

Line 6 — Construct an AS_0 variable definition for the parameters in hand and

Line 7 — Transform this variable definition.

Line 8 — Construct the *Qual* of the variable sort

Line 9 — Extract the parent in the case of a syntype

Line 10 — Construct the list (*tql*) of sort *Quals* corresponding to the variable list.

Line 11 — Construct the AS_1 identifier ($as_1 id$) of the parent partial type (tq') for the sort of the variable list (*nml*)

Line 12 — transform the name list to an AS_1 name list

Line 13 — The names in the list must be distinct

Line 16 — Construct the AS_1 formal parameter definition for the parameters in hand (*nml*).

Line 17 — Return the AS_1 formal parameter definition together with the rest of the formal parameters, return the associated sort list and return their *Dict* contributions

3.4.4 Signal Definitions

transform-signaldef(**mk-***Sigdef*$_0$(*elemlist*))(*dict*) $\triangleq$ (3.4.4.1)

```
1  (let mk-Sigelem0(nm, tidl, refinement) = hd elemlist in
2   let dlev = dict + [SCOPEUNIT ↦ dict(SCOPEUNIT) ⌢ ⟨(SIGNAL, nm)⟩] in
3   let (as1 refinement, subsig1, subsig2, d) =
4       if refinement = nil then
5         (nil, {}, {}, [])
6        else
7         (let mk-Refinement0(subsiglist) = refinement in
8          let (as1 set, s1, s2, d') = transform-refinement(subsiglist)(dlev) in
9          (mk-Signal-refinement1(as1 set), s1, s2, d')) in
10  let quall = ⟨get-visible-qual(tidl[i], TYPE)(dict) | 1 ≤ i ≤ len tidl⟩ in
11  let d' = [dict(SCOPEUNIT) ⌢ ⟨(SIGNAL, nm)⟩ ↦ mk-SignalD(quall, subsig1, subsig2)] in
12  let (as1 set', d'') =
13      if tl elemlist = ⟨⟩
14        then ({}, [])
15        else transform-signaldef(mk-Sigdef0(tl elemlist))(dict) in
16  if dom d' ∩ dom d'' ≠ {} then
17    exit("§2.2.2: Two definitions in the same scopeunit use the same name")
18   else
19    (let as1 idlist = ⟨make-as1-identifier(quall[i])(dict) | i ∈ ind quall⟩ in
20     let as1 tree = mk-Signal-definition1(name-to-name1(nm), as1 idlist, as1 refinement) in
21     ({as1 tree} ∪ as1 set', d + d' + d'')))
```

type: *Sigdef*$_0$ → *Dict* → *Signal-definition*$_1$-**set** *Dict*

Objective Transform a signal definition into AS_1

Parameters The composite AS_0 signal definition introducing a list (*elemlist*) of elementary signal definitions.

Result See *transform-decllist*

Algorithm

Line 1	Decompose the first signal definition in the composite AS_0 signal definition
Line 2	Update level to indicate the signal structure.
Line 3-9	Transform the refinement part of the signal definition and return the contained AS_1 definitions (*as*$_1$ *refinement*), the sub-signals leading in the same direction as the signal being defined (*subsig1*), the sub-signals leading in the opposite direction of the signal being defined (*subsig2*) and the *Dict* contribution from the new definitions (*d*).
Line 10	Construct the sort *Quals* from the specified AS_0 sort identifier list
Line 11	Construct the *Dict* contribution of the signal.
Line 12-15	Transform the rest of the signal definitions in the composite signal definition.
Line 16	No two signals in the composite definition may have the same name
Line 19	Construct the AS_1 sort identifier list.
Line 20-21	Return the set of AS_1 signal definitions and the *Dict* contributions from the signals.

transform-refinement(*decllist*)(*dict*) $\triangleq$ (3.4.4.2)

```
1  (if decllist = ⟨⟩ then
2     ({}, {}, {}, [])
3   else
4    (let (as1 declrest, sig1rest, sig2rest, drest) = transform-refinement(tl decllist)(dict) in
5     let mk-Subsignal0(reverse, sigdef) = hd decllist in
6     let (as1 decl, d) = transform-signaldef(sigdef)(dict) in
7     if dom d ∩ dom drest ≠ {} then
8       exit("§2.2.2: Two definitions in the same scopeunit and same entity class define the same name")
9      else
10      (let as1 tot = as1 declrest ∪ {mk-Subsignal-definition1(reverse, decl) | decl ∈ as1 decl} in
11       if reverse = nil then
12         (as1 tot, sig1rest ∪ dom d, sig2rest, d + drest)
13        else
14         (as1 tot, sig1rest, sig2rest + dom d, d + drest))))
```

type : *Subsignal*$_0^*$ $\rightarrow$ *Dict* $\rightarrow$ *Subsignal-definition*$_1$-**set** *Signalqual*-**set** *Signalqual*-**set** *Dict*

Objective Transform the sub-signals in a signal refinement into AS_1 subsignal definitions.

Parameters

decllist The list of AS_0 subsignal definitions

Result A set of AS_1 subsignal definitions, the set of signal *Quals* containing the signals leading in the same direction as the parent signal, the set of signal *Quals* containing the signals leading in the opposite direction as the parent signal (these two sets are in the function *transform-signaldef* put into the descriptor for the parent signal) and the *Dict* contribution for the sub-signals.

Algorithm

Line 1 When through then return no contributions.

Line 4 Transform the rest of the sub-signals.

Line 5 Decompose the first subsignal in the list.

Line 6 Transform the subsignal as if it was an ordinary signal.

Line 7 Check that the subsignal definitions are unique.

Line 10 Construct the set of AS_1 subsignal definitions from the AS_1 signal definitions and

Line 11-14 Return this set and if REVERSE was specified then add the signal *Quals* (taken from the domain of *d*) to the first *Signalqual* set, otherwise add them to the second *Signalqual* set. Also return the dict contributions for the sub-signals (*d*) and the contribution for the rest of the sub-signals (*drest*).

3.4.5 Procedure Definitions

transform-proceduredef(mk-*Procdef*$_0$(*pid*, *pl*, *decll*, *body*, *tid*))(*dict*) $\triangleq$ (3.4.5.1)

```
1  (let mk-Id0(q, name) = pid in
2   if q = ⟨⟩ ∧ tid ∈ {pid, nil} then
3     (let name' = if is-ServiceD(dict(dict(SCOPEUNIT))) then
4                     create-unique-name()
5                  else
6                     name in
7      let nqual = dict(SCOPEUNIT) ⌢ ⟨(PROCEDURE, name')⟩ in
8      let d = [SCOPEUNIT    ↦ dict(SCOPEUNIT) ⌢ ⟨(PROCEDURE, name)⟩,
```

```
 9              DATATYPEDEF ↦ initialdatadef(dict)] in
10     let (as1 list, pd, dict') = transform-procedureparml(pl)(dict + d),
11         (as1 declset, ddict) = transform-decllist(decll)(dict + d),
12         (mk-Procedure-graph1(sta, stateset), bdict) = transform-body(body)(dict + d) in
13     if dom dict' ∩ dom ddict ≠ {} then
14       exit("§2.2.2: Names of formal parameters must be distinct from variable names")
15       else
16       if (∃squal ∈ dom dict)(is-ServiceD(dict(squal)) ∧
17                              get-sur(squal) = process-or-service-level(dict(SCOPEUNIT))) ∧
18         stateset ≠ {}
19         then exit("§4.10.2: Procedure in decomposition or service has states or import expressions")
20         else (let as1 tree =
21                 make-as1 tree(PROCEDURE, name', as1 declset,
22                               as1 list, mk-Procedure-graph1(sta, stateset), nil)(ddict) in
23               let delem = [d(SCOPEUNIT) ↦ mk-ProcedureD(pd, nqual)] + dict' + ddict in
24               ({as1 tree}, delem + bdict)))
25   else
26    (q ≠ ⟨⟩
27       → exit("§2.4.1: Defining names may only be qualified in remote definitions"),
28     T → exit("§2.2.2: Ending name in procedure definition is different from defining name")))
```

type: $Procdef_0 \rightarrow Dict \rightarrow Procedure\text{-}definition_1\text{-set}\ Dict$

Objective Transform a procedure definition into AS_1

Parameters The procedure definition containing

pid	The procedure identifier (must contain a name only)
pl	The list of formal parameters
decll	The list of definitions
body	The state body
tid	The identifier (name) ending the definition

Result An AS_1 definition set consisting of one definition only and the *dict* contributions for the procedure and for the contained definitions

Algorithm

Line 1	Let *name* denote the procedure name
Line 2	The procedure name (identifier) must not be qualified (as it is not a remote definition) and the name (identifier) ending the procedure definition must be equal to the procedure name if specified.
Line 3	Construct a unique name for the procedure definition if it occurs in a service.
Line 7	Construct the variable *Qual* to be used in AS_1
Line 8-9	Update SCOPEUNIT to denote the procedure scopeunit and construct the initial $Data\text{-}type\text{-}definition_1$ for the procedure.
Line 10	Transform the formal procedure parameters in order to obtain the AS_1 parameters ($as_1 list$), their sort descriptors (to be used when the procedure is invoked), and their *Dict* contributions.
Line 11	Transform the contained definitions
Line 12	Transform the procedure body and return the AS_1 procedure graph and a *Dict* wherein only the IMPLIED entry and the OUTSIGNALS entry are used.
Line 13	No formal parameter may have the same name as a variable or synonym defined in the procedure
Line 16-19	If a procedure containing states or imports is defined in a process or in a procedure in a process then the process must not contain services.

Line 20-24 Return the AS_1 procedure definition ($as_1 tree$) and a *Dict* contribution consisting of the procedure descriptor the formal parameter descriptors, and the descriptors of the entities defined in the procedure (*bdict*).

transform-procedureparml(*pl*)(*dict*) $\triangleq$ (3.4.5.2)

```
1  (if pl = ⟨⟩ then
2     (⟨⟩, ⟨⟩, [])
3   else
4    (let (as1 elems, tpelems, delems) =
5          cases hd pl:
6           (mk-Inoutparm0(varlist, tid)
7              → transform-varparm(true, varlist, tid)(dict),
8            mk-Inparm0(varlist, tid)
9              → transform-varparm(false, varlist, tid)(dict)) in
10   let (as1 list, tplist, d) = transform-procedureparml(tl pl)(dict) in
11   if dom delems ∩ dom d ≠ {} then
12     exit("§2.2.2: Two definitions in the same scopeunit use the same name")
13    else
14     (as1 elems ⌢ as1 list, tpelems ⌢ tplist, delems + d)))
```

type: $Procparm_0^* \rightarrow Dict \rightarrow Procedure\text{-}formal\text{-}parameter_1^* \; FormparmD^* \; Dict$

Objective Transform a list of formal procedure parameters into AS_1

Parameters

pl The list of formal AS_0 parameters.

Result The AS_1 Formal parameters, the descriptors (*FormparmD**) which are put into the *ProcedureD* descriptor and used for type checking of the actual parameters and finally the *Dict* contributions for the parameters.

Algorithm

Line 1 When through, return empty lists and the empty *Dict*

Line 4-9 Transform the first parameter which is either an INOUT variable parameter (line 6) or an IN variable parameter (line 8).

Line 10 Transform the rest of the formal parameter list

Line 11 The names introduced in *pl* must be distinct

Line 14 Return the objects composed of the first formal parameter and the rest.

transform-varparm(*isout*, *varlist*, *tid*)(*dict*) $\triangleq$ (3.4.5.3)

```
 1  if card elems varlist ≠ len varlist then
 2    exit("§2.2.2: Two definitions in the same scopeunit use the same name")
 3   else
 4    (let tqual = get-visible-qual(tid, TYPE)(dict) in
 5     let tqual' = get-parent(tqual)(dict) in
 6     let as1 varlist = ⟨name-to-name1(varlist[i]) | 1 ≤ i ≤ len varlist⟩ in
 7     let as1 tree = if isout then
 8                      mk-Inout-parameter1(as1 varlist, make-as1-identifier(tqual)(dict))
 9                    else
10                     mk-In-parameter1(as1 varlist, make-as1-identifier(tqual)(dict)),
11         parmdescrlist = if isout then
12                           ⟨mk-InoutDescr(tqual) | 1 ≤ i ≤ len varlist⟩
13                         else
14                           ⟨mk-InDescr(tqual') | 1 ≤ i ≤ len varlist⟩ in
15     let nmlist =
16         ⟨if is-ServiceD(dict(dict(SCOPEUNIT)))
17            then create-unique-name()
18            else varlist[i] | 1 ≤ i ≤ len varlist⟩,
19         newquallist = ⟨dict(SCOPEUNIT) ⌢ ⟨(VALUE, nmlist[i])⟩ | 1 ≤ i ≤ len varlist⟩ in
20     let vquall = ⟨dict(SCOPEUNIT) ⌢ ⟨(VALUE, varlist[i])⟩ | 1 ≤ i ≤ len varlist⟩,
21         delem = [vquall[i] ↦ mk-VarD(tqual, nil, nil, nil, newquallist[i]) | 1 ≤ i ≤ len varlist] in
22     (⟨as1 tree⟩, parmdescrlist, delem))
```

type: *Bool* $Name_0^*$ $Id_0 \rightarrow$ *Dict* $\rightarrow$ *Procedure-formal-parameter*$_1^+$ *FormparmD*$^+$ *Dict*

Objective Transform a formal parameter into AS_1

Parameters

isout	True if the formal parameter is an INOUT parameter.
varlist	The names of the formal parameters
tid	Their sort.

Result See *transform-procedureparml*

Algorithm

Line 1	The names in the formal parameters must be distinct
Line 4	Extract the *Qual* of the parameter sort.
Line 5	Extract the parent partial type of the sort.
Line 6	Transform the name list into AS_1 names
Line 7-10	Construct the AS_1 formal parameter.
Line 11-14	Construct the actual parameter sort descriptor.
Line 21	Construct the dict contribution for the formal parameter. The contained descriptors are *VarD* descriptors, (like for ordinary variable) since formal variable parameters acts as ordinary variables with respect to the static properties.

3.4.6 Sort Generators

transform-sortgenerator(**mk-***Sortgenerator*$_0$(*nm*, *parml*, *geninstl*, *prop*, *tnm*))(*dict*) $\triangleq$ **(3.4.6.1)**

```
1  (if tnm ∈ {nm, nil} ∧ gen-formparm-unique(parml, {nm}, {}, {}) then
2     (let gqual = dict(SCOPEUNIT) ↷ ⟨(GENERATOR, nm)⟩ in
3      let prop' = transform-geninst(nm, geninstl, prop)(dict + [gqual ↦ mk-ErrorD()]) in
4      let descr = mk-GeneratorD(parml, prop') in
5      ({}, [gqual ↦ descr]))
6    else
7     exit("§5.4.1.12: Ending name in generator definition is different from defining name"))
```

type : *Sortgenerator*$_0$ → *Dict* → *Decl*$_1$-**set** *Dict*

Objective Construct the *Dict* contribution corresponding to a data sort generator. No AS_1 objects are returned.

Parameters An AS_0 data sort generator containing

nm	The generator name
parml	The formal parameters
geninstl	A list of contained generator instantiations
prop	The normal properties
tnm	The name ending the definition

Result For convenience the function returns a set of AS_1 definitions though the list is always empty (see *transform-decllist*). Also returned is the *Dict* contribution of the data sort generator.

Algorithm

Line 1	The name ending the definition must be equal to the generator name if specified and the formal parameters within the various parameter classes must be unique.
Line 2	Construct the *Qual* for the generator.
Line 3	Transform the generator instantiations contained in the generator definition. The generator qual (*gqual*) is denoted by a *ErrorD* such that recursion can be detected.
Line 4-7	Return the *Dict* contribution containing the generator formal parameters (*parml*), and the generator body (*prop*). Well-formedness is applied to the body after it is replaced by the generator instance construct (not at this place.)

gen-formparm-unique(*parml*, *sset*, *lset*, *oset*) $\triangleq$ **(3.4.6.2)**

```
1  if parml = ⟨⟩ then
2    true
3   else
4    cases hd parml:
5     (mk-Sortparm0(nmlist)
6        → card elems nmlist = len nmlist ∧ elems nmlist ∩ sset = {} ∧
7          gen-formparm-unique(tl parml, sset ∪ elems nmlist, lset, oset),
8      mk-Termparm0(nmlist),
9      mk-Litparm0(nmlist)
10       → card elems nmlist = len nmlist ∧ elems nmlist ∩ lset = {} ∧
11         gen-formparm-unique(tl parml, sset, lset ∪ elems nmlist, oset),
12     mk-Opparm0(nmlist)
13       → card elems nmlist = len nmlist ∧ elems nmlist ∩ oset = {} ∧
14         gen-formparm-unique(tl parml, sset, lset, oset ∪ elems nmlist))
```

type : *Genparm*$_0$ *Name*$_0$-**set** *Name*$_0$-**set** *Name*$_0$-**set*** → *Bool*

Objective Check that the formal parameters for a generator are unique.

Parameters

parml	The formal parameters
sset,lset,oset	The set of sort parameters, literal or term parameters and operator parameters respectively considered so far (the function is recursive).

Result **true** if success

Algorithm

Line 5-7	For a sort parameter, the contained names must be unique and none of the names may occur in another sort parameter and the rest of formal generator parameters must also be unique.
Line 8-11	For a term parameter or a literal parameter, the contained names must be unique and none of the names may occur in another term or literal parameter and the rest of formal generator parameters must also be unique.
Line 12-14	For an operator parameter, the contained names must be unique and none of the names may occur in another operator parameter and the rest of formal generator parameters must also be unique.

3.4.7 Sort Definitions

transform-partial-typedef(mk-*Partialtypedef*$_0$(*nm*, *extprop*, *prop*, *vlist*, *tnm*))(*dict*) $\triangleq$ (3.4.7.1)

```
1  (if tnm ∈ {nm, nil} then
2     (let tqual = dict(SCOPEUNIT) ⌢ ⟨(TYPE, nm)⟩ in
3      let (implicitdecl, delem) = make-implicit-sigdecls(tqual)(dict) in
4      if (∃opqual ∈ dom dict)((is-OperatorD(dict(opqual)) ∨ is-LiteralD(dict(opqual))) ∧
5                              s-Result(dict(opqual)) = tqual ∧
6                              get-sur(get-sur(opqual)) = dict(SCOPEUNIT)) then
7        if vlist = nil then
8          (let dict' = cases extprop:
9                       (nil
10                         → transform-sortdef(nm, prop, nil)(dict),
11                      mk-Struc0()
12                         → transform-struc(nm, extprop, prop)(dict),
13                      mk-Inherited0(, , )
14                         → transform-inherited(nm, extprop, prop)(dict),
15                      mk-Geninstlist0(instl)
16                         → (let prop' = transform-geninst(nm, instl, prop)(dict) in
17                            transform-sortdef(nm, prop', nil)(dict))) in
18           (implicitdecl, dict'))
19         else
20          (let unm = create-unique-name() in
21           let (, dnew) =
22               transform-partial-typedef(mk-Partialtypedef0(unm, extprop, prop, nil, unm))(dict),
23               id = mk-Id0(dict(SCOPEUNIT), unm) in
24           let (as1 syn, dsyn) =
25               transform-syntype(mk-Syntypedef0(nm, id, vlist, nil, tnm))(dict) in
26           (implicitdecl ∪ as1 syn, delem + dnew + dsyn))
27       else
28        exit("§5.2.1: There exist no operators which returns a value of that sort"))
29    else
30     exit("§5.2.1: Ending name in partial type definition is different from defining name"))
```

type: *Partialtypedef*$_0$ → *Dict* → *Decl*$_1$-**set** *Dict*

Objective Transform a partial data type definition into AS_1

Parameters The data sort definition containing

nm	The sort name
extprop	The extended properties
prop	The properties (literals, operators, equations etc.)
vlist	A value list in the case of a syntype
tnm	The name ending the definition

Result See *transform-decllist*

Algorithm

Line 1	The name ending the data sort definition must be equal to the data sort name if specified.
Line 2	Construct the *Qual* representing the sort identifier.
Line 3	Construct the AS_1 definitions which defines the implicit signals attached to all the IMPORT - EXPORT variables of the sort being defined.
Line 4	There must exist an operator or literal which
Line 5	has this sort as result sort and which
Line 6	is defined in the surrounding scopeunit
Line 7	If the CONSTANTS construct is omitted then it is a real sort definition
Line 8	Construct the *Dict* contribution containing the sort descriptor, the operator and literal descriptors and the updated *Data-type-definition*$_1$. The AS_0 sort definition is either
Line 9	A (simple) sort definition (with no extended properties).
Line 11	A struct definition or
Line 13	A sort which is based on inheritance or
Line 15	A (sequence of) generator instance(s). In this case, the instances are expanded (line 15) before they are transformed in the ordinary way (line 17)
Line 20-26	If CONSTANTS is specified then it is a syntype definition which means that
Line 20	A unique implicit sort name is generated and
Line 21	The *Partialtypedef*$_0$ without any CONSTANTS and with that new name is transformed and
Line 24	The *Syntypedef*$_0$ with CONSTANTS (*vallist*) and the implicit sort as parent is transformed

make-implicit-sigdecls(*tqual*)(*dict*) $\triangleq$ (3.4.7.2)

1 (**let** *qset* = $\{(tq,) \in$ **dom** *dict* | *tq* = *tqual*$\}$ **in**
2 *transform-decllist*($\{$*make-implicit-decl*(*tqual*, *dict*(*t*)) | *t* $\in$ *qset*$\}$)(*dict*))

type: *Sortqual* $\rightarrow$ *Dict* $\rightarrow$ *Decl*$_1$-**set** *Dict*

Objective Construct the AS_1 signal definitions attached to all the *NameclosureD*s which contains the sort *tqual* (see the definition of *Exportmap*).

Algorithm

Line 1	Construct the set of those *NameclosureD*s of which the *Sortqual* equals *tqual* and
Line 2	Construct the list of AS_0 definitions constructed by joining the lists of definitions attached to every element in the set and transform this list into AS_1.

make-implicit-decl(*tqual*, mk-*SignalnamesD*(, *xqnm*, *xrnm*)) ≜ (3.4.7.3)

```
1  (let as0tid = as0-id(tqual) in
2   mk-Sigdef0(⟨mk-Sigelem0(xqnm, ⟨⟩, nil),
3              mk-Sigelem0(xrnm, ⟨as0tid⟩, nil)⟩))
```

type: *Sortqual SignalnamesD* → *Sigdef*$_0$

Objective Construct the AS$_0$ signal definitions attached to EXPORT - IMPORT of a certain name and a certain sort. (*tqual*).

Algorithm

line 1	Construct the AS$_0$ identifier of the sort carried by the xtREPLY signal
Line 2-3	Construct the signal definition containing
Line 2	The xtQUERY signal and
Line 3	The xtREPLY signal.

transform-inherited(*nm*, mk-*Inherited*$_0$(*pid*, *lrenam*, *ops*), *prop*)(*dict*) ≜ (3.4.7.4)

```
1  (let pqual = get-inherited-parent(get-visible-qual(pid, TYPE)(dict))(dict) in
2   if is-recursive-sort(pqual, {})(dict) then
3     exit("§5.4.1.11: Sort is based on itself")
4    else
5     (let pqual' = s-Newqual(dict(pqual)) in
6      let sortdict = transform-sortdef(nm, prop, pqual)(dict) in
7      let typedef = sortdict(DATATYPEDEF) in
8      let mk-Data-type-definition1(typename, union, sorts, sigs, eqs) = typedef in
9      let iqual be s.t. iqual ∈ dom sortdict ∧ is-SortD(sortdict(iqual)) in
10     let mk-SortD(eqs', pq, exp1, nqual) = sortdict(iqual) in
11     let (litmap, litd) = transform-literal-renaming(lrenam, iqual, pqual)(dict) in
12     let concaxioms1 = make-as1-concaxioms(dom litd)(dict) in
13     let (opmap, opd, opset) = transform-operator-renaming(ops, iqual, pqual)(dict) in
14     if dom sortdict ∩ dom litd ∩ dom opd ≠ {} then
15       exit("§5.2.2: Operator or literal both defined explicit and by inheritance")
16      else
17       (let inhax = extract-inherited-axioms(litmap, opmap, nqual, pqual')(dict) in
18        let iid1 = make-as1-identifier(nqual)(dict) in
19        let litsig = {mk-Literal-signature1(nm1, iid1) | mk-Identifier1(, nm1) ∈ rng litmap} in
20        let datatypedef' =
21            mk-Data-type-definition1(typename, union, sorts,
22                                     sigs ∪ litsig ∪ opset, eqs ∪ inhax ∪ concaxioms1) in
23        let sortdescr = mk-SortD(eqs' ∪ inhax ∪ concaxioms1, pq, exp1, nqual) in
24        (sortdict + litd + opd + [iqual ↦ sortdescr,
25                                  DATATYPEDEF ↦ datatypedef'])))))
```

type: *Name*$_0$ *Inherited*$_0$ *Properties*$_0$ → *Dict* → *Dict*

Objective Transform a partial type definition which is based on inheriting.

Parameters

nm	The name of the sort being defined
Inherited	The AS$_0$ inherit construct
prop	The AS$_0$ properties defined in the ADDING construct.

Result The *Dict* where the DATATYPEDEF entry has been updated to include the sort being defined and where descriptors for the sort and operators has been added.

Algorithm

Line 1	Extract the *Qual* denoting the sort identifier of the parent.
Line 2-3	The sort must not inherit from itself
Line 5	Extract the unique *Qual* of the parent
Line 6	Update the DATATYPEDEF entry with the sort being defined and with its ADDING properties.
Line 8	Decompose the updated *Data-type-definition*$_1$.
Line 9-10	Extract the (just added) *Qual* of the sort being defined and decompose it.
Line 11	Transform the literal renaming part into a map *litmap* and construct descriptors for the literal names introduced in the literal renaming.
Line 12	Construct the axioms implied from any character string literals introduced in the literal renaming.
Line 13	Transform the operator renaming part into a map *opmap* and construct descriptors for the new operator names (*opd*) and construct the AS_1 signatures for the operators (*opset*). This function also deals with the operators which are implicitly inherited (the invisible operators)
Line 14-15	all operators and literals defined in the adding part *sortdict* and the operators (*opd*) and literals (*litd*) defined in the inheritance part must have different signatures.
Line 17	Extract all the axioms using the operators which are implicitly inherited.
Line 18	Construct the AS_1 identifier of the sort being defined.
Line 19	Construct the AS_1 signatures of the renamed literals.
Line 20-23	Include the properties from the inheritance part in the *Data-type-definition* and in the descriptor for the sort.
Line 24	Return the *Dict* contribution from the adding part (*sortdict*), for the renamed literals (*litd*), for the renamed operators (*opd*), for the sort descriptor and the updated *Data-type-definition*$_1$

is-recursive-sort(*qual*, *qset*)(*dict*) $\triangleq$ (3.4.7.5)

```
1  (if qual ∈ qset then
2     true
3   else
4     cases dict(qual):
5      (mk-SortD(, pa, , )
6         → if pa = nil then false else is-recursive-sort(pa, qset ∪ {qual})(dict),
7       mk-SyntypeD(pa, , , )
8         → is-recursive-sort(pa, qset ∪ {qual})(dict)))
```

type: *Sortqual Sortqual*-set → *Dict* → *Bool*

Objective Check that a syntype or an inheriting sort is not based on itself. The function traverse recursively through the *dict* until a partial type descriptor is found which is not based in inheritance. (I.e. until the false condition applies).

Parameters

qual	Denotes the sort in hand
qset	Denotes the set of sorts which already have been referred. When the function initially is applied, this set is empty.

Result True if the sort is recursively defined.

Algorithm

Line 1	If the sort in hand (*qual*) already has been referred then **true** else
Line 5-6	if the sort in hand is a partial type having no parent then the sort is not recursively defined, otherwise add the partial type to the sortset and proceed with its parent.
Line 7	Add the syntype in hand to the sortset and proceed with its parent.

transform-literal-renaming(*lrenam*, *qual*, *pqual*)(*dict*) $\triangleq$ **(3.4.7.6)**

```
1  (let plitset = {lq ∈ dom dict | is-LiteralD(dict(lq)) ∧ s-Sortqual(dict(lq)) = pqual} in
2   let defaultmap = [qual ↦ qual | qual ∈ plitset] in
3   let litmap = defaultmap + build-literal-renaming(lrenam, qual, pqual, plitset) in
4   ([make-as1-identifier(lq)(dict) ↦ make-as1-identifier(litmap(lq))(dict) | lq ∈ dom litmap],
5    [lq ↦ mk-LiteralD(qual) | lq ∈ rng litmap]))
```

type : $Literalrenaming_0$ *Sortqual Sortqual* $\rightarrow$ *Dict* $\rightarrow$
($Literal\text{-}operator\text{-}identifier_1$ $\overset{m}{\rightarrow}$ $Literal\text{-}operator\text{-}identifier_1$) *Dict*

Objective Construct a map from parent literals into new literals to be used when the inherited axioms are to be extracted (in *extract-inherited-axioms*) and construct *Dict* descriptors for the new literals

Parameters

lrenam	The AS_0 literal renaming
qual	The *Qual* of the sort being defined
pqual	The *Qual* of the sort on which the inheritance is based.

Result The constructed map and the *Dict* contribution containing the literal descriptors

Algorithm

Line 1	Extract from *Dict*, the set of literals defined for the parent sort.
Line 2	Construct the default map, i.e. the corelation which applies for the literals which are not mentioned in the renaming part.
Line 3	Construct literal renaming map (where the contained literals are *Quals*) by overwriting the default map.
Line 4	Return the literal renaming map after having converted the *Quals* to AS_1 identifiers and return
Line 5	The literal descriptors (*LiteralDs*) which all contain the *Qual* of their sort

build-literal-renaming(*lrenam*, *qual*, *pqual*, *plitset*) $\triangleq$ (3.4.7.7)

```
 1  if lrenam = ⟨⟩ then
 2    []
 3   else
 4    (let mk-Literalpair0(nlit, olit) = hd lrenam in
 5     let nqual = qual ⌢ ⟨(LITERAL, nlit)⟩,
 6         oqual = pqual ⌢ ⟨(LITERAL, olit)⟩ in
 7     if oqual ∈ plitset then
 8       (let restmap = build-literal-renaming(tl lrenam, qual, pqual, plitset) in
 9        if nqual ∈ rng restmap then
10          exit("§5.4.1.11: Literal defined twice in renaming")
11         else
12          if oqual ∈ dom restmap then
13            exit("§5.4.1.11: Literal renamed twice")
14           else
15            restmap + [oqual ↦ nqual])
16      else
17       exit("§5.4.1.11: Literal in literal renaming is not defined in the parent sort"))
```

type : *Literalrenaming*$_0$ *Sortqual Sortqual Qual*-**set** $\rightarrow$ (*Qual* $\overset{m}{\rightarrow}$ *Qual*)

Objective Construct from the AS_1 renaming part a map from parent *Quals* into *Quals* of the new literal names.

Parameters

lrenam	The AS_0 literal renaming
qual	The *Qual* of the sort being defined
pqual	The *Qual* of the sort on which the inheritance is based
plitset	The set of *Quals* containing all the literals defined for the parent sort

Result The constructed literal map

Algorithm

Line 1	When through then return nothing (the function is recursive).
Line 4	Decompose the next pair of literal names in the literal renaming.
Line 5-6	Construct the *Quals* of the new literal name (*nqual*) and of the parent (old) literal name (*oqual*).
Line 7	The right-hand literal in *Literalpair*$_0$ must be a literal of the parent sort.
Line 8	Construct the map for the rest of the literal renaming.
Line 9-12	If the lefthand literal (line 9) or the right-hand literal (line 12) is in the map for the rest of the literal renaming then it is mentioned twice in the literal renaming
Line 15	Else return the literal map for the rest of the literal renaming where this contribution has been added

transform-operator-renaming(*oprenam*, *iqual*, *pqual*)(*dict*) $\triangleq$ (3.4.7.8)

```
1  (let opset = {qual ∈ dom dict | is-OperatorD(dict(qual)) ∧
2                                  (let mk-OperatorD(argl, res, , ) = dict(qual) in
3                                   pqual ∈ elems argl ∨ pqual = res)} in
4   let explicitset = {qual ∈ opset | s-Explicit(dict(qual))} in
5   let allnameset = {name | (∃q ∈ explicitset)((, (name, , )) = q[len q])} in
6   let oprenam' = if oprenam = ALL then ⟨mk-Operatorpair0(nil, nm) | nm ∈ allnameset⟩ else oprenam in
7   let (opmap, opd) = build-operator-renaming(oprenam', iqual, pqual, explicitset)(dict) in
8   let (opmap', opd') = build-implicit-operators(iqual, pqual, opset \ explicitset)(dict) in
9   let (qmap, opdict) = (opmap + opmap', opd + opd') in
10  let idmap = [make-as1-identifier(qual)(dict) ↦ make-as1-identifier(qmap(qual))(dict) |
11               qual ∈ dom qmap] in
12  let as1typing = {make-as1-typing(op)(dict) | op ∈ dom opdict} in
13  (idmap, opdict, as1typing))
```

type: (ALL | *Operatorrenaming*$_0$) *Sortqual Sortqual* $\rightarrow$ *Dict* $\rightarrow$
(*Operator-identifier*$_1$ $\overset{m}{\rightarrow}$*Operator-identifier*$_1$) *Dict Operator-signature*$_1$-**set**

Objective Construct from the AS_1 operator renaming part a map from parent operator identifiers into operator identifiers of the sort being defined. The map includes the identifiers of the operators which are implicitly inherited (an operator is implicitly inherited if it includes the parent sort in its signature)

Parameters

oprenam	The AS_0 operator renaming
iqual	The *Qual* of the sort being defined
pqual	The *Qual* of the parent

Result The constructed operator map

Algorithm

Line 1	Construct the set of operator *Qual*s from *Dict* for which it holds that the parent *Qual* (*pqual*) occur in the argument sort list (*argl*) or is the result of the operator.
Line 4	Construct the subset of *opset* which contains those operators which are explicitly defined (or inherited).
Line 5	Construct the set of operator names for the explicit defined operators.
Line 6	If ALL is specified then let *oprenam'* denote the *Operatorrenaming*$_0$ which includes all explicit operators else let *oprenam'* be *oprenam*.
Line 7	Construct the operator *Qual* map and the *Dict* descriptors for the explicit operators.
Line 8	Construct the operator *Qual* map and the *Dict* descriptors for the implicit operators.
Line 9	The complete operator *Qual* map is *qmap* and the complete *Dict* contribution is *opdict*.
Line 10	Convert the operator *Qual* map to an operator identifier map.
Line 12	For each operator in the *Dict* contribution, construct the AS_1 operator signature.
Line 13	Return the operator identifier map, the *Dict* contributions and the AS_1 signatures of the operators

make-as$_1$*-typing*(*qual*)(*dict*) $\triangleq$ (3.4.7.9)

```
1  (let mk-OperatorD(sl, res, qual', ) = dict(qual) in
2   let (, (nm, , )) = qual[len qual'] in
3   let sl1 = ⟨make-as1-identifier(sl[i])(dict) | 1 ≤ i ≤ len sl⟩,
4       res1 = make-as1-identifier(res)(dict) in
5   mk-Operator-signature1(name-to-name1(nm), sl1, res1))
```

type : *OperatorD* → *Dict* → *Operator-signature*$_1$

Objective From an operator *Qual* construct the AS_1 signature of the operator

Parameters

qual The operator *Qual*

Result The constructed operator signature

Algorithm

Line 1 Decompose the operator descriptor. *qual'* denotes the *Newqual* containing the name to be used in AS_1

Line 2 Extract the operator name by decomposing the last element in the *Qual*. For an operator, the last element always is an *Operatorqualelem* (see the domain).

Line 3 Construct the AS_1 identifiers list denoting the argument sorts.

Line 4 Construct the AS_1 identifier denoting the result sort.

Line 5 Return the composed operator signature

build-operator-renaming(*renlist*, *qual*, *pqual*, *opset*)(*dict*) $\triangleq$ (3.4.7.10)

```
1  (if renlist = ⟨⟩ then
2     ([], [])
3   else
4    (let mk-Operatorpair0(nop, oop) = hd renlist in
5     let oset = {oqual ∈ opset | (let (, (nm, , )) = oqual[len oqual] in
6                                  get-sur(oqual) = pqual ∧
7                                  nm = oop)} in
8     if oset = {} ∨ (is-Name0(oop) ∧ (let mk-Name0(, exc) = oop in
9                                      exc ≠ nil)) then
10      if oset = {} then
11        exit("§5.4.1.11: Operator in operator renaming is not defined in the parent sort")
12       else
13        exit("§5.4.1.11: Operator with an exclamation is mentioned in an operator renaming")
14      else
15       (let (maprest, drest) =
16            build-operator-renaming(tl renlist, qual, pqual, opset \ oset)(dict) in
17        let (map, d) = rename-operator(qual, pqual, oset, nop)(dict) in
18        if dom d ∩ dom drest = {} then
19          (map + maprest, d + drest)
20         else
21          exit("§5.4.1.11: Operator renamed twice"))))
```

type : *Operatorrenaming*$_0$ *Sortqual Sortqual Qual*-set → *Dict* → (*Qual* $\overset{m}{\rightarrow}$*Qual*) *Dict*

Objective From an operator renaming, construct a map from parent operator *Qual*s into operator *Qual*s of the sort being defined. Also construct the *Dict* contributions for the new operators

Parameters

renlist The AS_0 operator renaming list

qual	The *Qual* of the sort being defined
pqual	The *Qual* of the parent sort
opset	The set containing all *Quals* of the explicitly defined operators

Result The constructed operator *Qual* map and the *Dict* contributions

Algorithm

Line 1	When through, return nothing (the function is recursive).
Line 4	Decompose the next rename pair in the rename list.
Line 5	Extract the set of operator *Quals* for which the surrounding scope-unit of the place where it is defined is the parent *Qual* and the name part is equal to the parent operator name mentioned in the renaming.
Line 8-11	There must exist at least one *Qual* in the set.
Line 8-13	The parent operator name must not contain an exclamation mark.
Line 15	Construct the operator map and the *Dict* contributions for the rest of the renaming list.
Line 17	Rename the operator in hand. *map* denotes the contributions to the operator map and *d* denotes the contribution to the *Dict*.
Line 18-19	If the *Quals* of the operator name in hand (there may be several *Quals* for the operator name because the parent operator name mentioned in the operator renaming may denote several (overloaded) operators) are not in the *Dict* contributions for the rest of the renaming list then return the operator map contributions for the operator name in hand (*map*) and for the rest of the list and return the corresponding *Dict* contributions

$rename\text{-}operator(iqual, pqual, oset, nop)(dict) \triangleq$ (3.4.7.11)

```
1  if oset = {} then
2    ([], [])
3   else
4    (let qual ∈ oset in
5     let (, (nm, , )) = qual[len qual] in
6     let mk-OperatorD(arglist, result, , ) = dict(qual) in
7     let arglist' = ⟨if get-parent(arglist[i])(dict) = pqual then iqual else arglist[i] | 1 ≤ i ≤ len arglist⟩ in
8     let parglist = ⟨get-parent(arglist'[i])(dict) | 1 ≤ i ≤ len arglist'⟩ in
9     let result' = if get-parent(result)(dict) = pqual then iqual else result in
10    let presult = get-parent(result')(dict) in
11    let (maprest, drest) = rename-operator(iqual, pqual, oset \ {qual}, nop)(dict) in
12    if result = result' ∧ arglist = arglist' then
13      (maprest, drest)
14     else
15      (let nm' = if nop = nil then nm else nop in
16       let newnm = create-unique-name() in
17       let opqual = iqual ⌢ ⟨(OPERATOR, (nm', parglist, presult))⟩ in
18       let newqual = iqual ⌢ ⟨(OPERATOR, (newnm, parglist, presult))⟩ in
19       (maprest + [iqual ↦ newqual], drest + [opqual ↦ mk-OperatorD(arglist', result', newqual, true)])))
```

type: $Sortqual\ Sortqual\ Qual\text{-set}\ Newoperator_0 \rightarrow Dict \rightarrow (Qual \xrightarrow{m} Qual)\ Dict$

Objective Construct a map from parent operator *Quals* into operator *Quals* for a given operator name. Also construct the *Dict* contributions for the operators denoted to the name

Parameters

iqual	The *Qual* of the inheriting sort
pqual	The *Qual* of the parent sort

oset	The set of operator *Quals* defined in the parent sort and having the name mentioned in the operator renaming
nop	The new operator name

Result The constructed map and the corresponding *Dict* contributions

Algorithm

Line 1	When through then return nothing (the function is recursive).
Line 4-5	Take the next operator *Qual* in the set and extract the operator name.
Line 6	Decompose the operator descriptor for the parent operator in hand
Line 7	Replace every occurrence of the parent sort in the argument sort list for the operator in hand by the inheriting sort. Note that the function *get-parent* has nothing to do with the parent of the inheriting sort. *get-parent* replaces any syntypes by its (parent) sort.
Line 8	Remove any syntypes from the argument list. This nominal list is used in the *Qual* for the operator.
Line 9-10	Do the same for the result sort.
Line 11	Go through the rest of the set of parent operator *Quals*.
Line 12-13	If the parent *Qual* is not used in the argument list or in the result of the operator then return the map and the *Dict* contributions for the rest of the list else
Line 15	If a new operator name is not specified in the $Operatorpair_0$ then the inherited operator has the same name as the parent operator.
Line 16	Construct a unique name to be used in the *Newqual* for the operator (see the definition of *OperatorD*).
Line 17	Construct the *Qual* to be used as the *Dict* entry for the operator.
Line 18	Construct the *Newqual* for the operator.
Line 19	Return the operator *Qual* map for the rest of the operators and for the operator in hand and return the *Dict* for the contributions

$build\text{-}implicit\text{-}operators(iqual, pqual, opset)(dict) \triangleq$ (3.4.7.12)

```
 1  if opset = {} then
 2    ([], [])
 3   else
 4    (let qual ∈ opset in
 5     let (, (, arglist, result)) = qual[len qual] in
 6     let arglist' = ⟨if get-parent(arglist[i])(dict) = pqual then iqual else arglist[i] | 1 ≤ i ≤ len arglist⟩ in
 7     let result' = if get-parent(result)(dict) = pqual then iqual else result in
 8     let (restmap, drest) = build-implicit-operators(iqual, pqual, opset \ {qual})(dict) in
 9     if arglist = arglist' ∧ result = result' then
10       (restmap, drest)
11      else
12       (let newnm = create-unique-name() in
13        let opqual = iqual ⌢ ⟨(OPERATOR, (newnm, arglist', result'))⟩ in
14        (restmap + [qual ↦ opqual], drest + [opqual ↦ mk-OperatorD(arglist', result', opqual, false)])))
```

type : *Sortqual Sortqual Qual*-set → *Dict* → (*Qual* $\overrightarrow{m}$ *Qual*) *Dict*

Objective Construct a map from parent operator *Quals* into operator *Quals* for the implicit operators. Also construct the *Dict* contributions for the operators.

Parameters

iqual	The *Qual* of the inheriting sort

pqual	The *Qual* of the parent sort
oset	The set of operator *Qual*s using the parent sort

Result The constructed map and the corresponding *Dict* contributions

Algorithm

Line 1	When through, return nothing (the function is recursive).
Line 4	Take an operator *Qual* in the set and extract the argument sort list and the result sort.
Line 6-7	Construct the argument sort list where every occurrence of the parent sort is replaced by the inheriting sort (*iqual*) and do the same for the result sort.
Line 8	Construct the *Qual* map and the *Dict* contributions for the rest of the operators in *opset*.
Line 9-10	If *pqual* is not used in the signature then return the constructed objects.
Line 12-13	Create a new unique name for the implicit operator and construct the *Qual* containing this name.
Line 14	Return the *Qual* map and the *Dict* contributions for the rest of the operators in *opset* and return the map contribution for the operator and its *Dict* contribution

extract-inherited-axioms(*litmap*, *opmap*, *qual*, *pqual*)(*dict*) $\triangleq$ **(3.4.7.13)**

```
1  (let qualset = all-visible-sorts(dict) \ {qual} in
2   let id1 = make-as1-identifier(qual)(dict) in
3   let pid1 = make-as1-identifier(pqual)(dict) in
4   let axset = union {s-Equations1(dict(qual)) | qual ∈ qualset} in
5   let axset' = {convert-axiom(axiom, litmap, opmap, id1, pid1) | axiom ∈ axset} in
6   axset' \ axset)
```

type: (*Literal-operator-identifier*$_1$ $\overset{m}{\rightarrow}$ *Literal-operator-identifier*$_1$)
(*Operator-identifier*$_1$ $\overset{m}{\rightarrow}$ *Operator-identifier*$_1$) *Sortqual Sortqual* $\rightarrow$
Dict $\rightarrow$ *Equations*$_1$

Objective Extract all the inherited axioms, that is, all the axioms which uses any of the operators or literals which includes the parent sort in the signature

Parameters

litmap	A map from parent literal identifiers into literal identifiers defined for an inheriting sort
opmap	A map from parent operator identifiers into operator identifiers defined (implicit or explicit) for an inheriting sort
qual	The *Qual* of the inheriting sort
pqual	The *Qual* of the parent sort

Result The set of equations using any of the parent operators/literals, where the parent operators/literals identifiers have been replaced by the operator/literal identifier of the inheriting sort

Algorithm

Line 1	Construct the set of sort *Qual*s which are visible in the enclosing scopeunit. Exclude the inheriting sort from the set.
Line 2-3	Construct the AS_1 identifiers of the inheriting sort and the parent sort respectively.
Line 4	Construct the set which is the union of the set of equations defined in the various (visible) sorts.

Line 5	Construct the set consisting of all the equations defined in a visible sort and such that each equation is converted to include the new operators/literals instead of the parent operators/literals.
Line 6	Return this set, but exclude those equations which has not changed

convert-axiom(*axiom*, *lmap*, *omap*, *id*, *pid*) $\triangleq$ (3.4.7.14)

```
1   cases axiom:
2    (mk-Unquantified-equation1(t1, t2)
3       → mk-Unquantified-equation1(convert-term(t1, lmap, omap), convert-term(t2, lmap, omap)),
4     mk-Quantified-equations1(nms, sort, eqs)
5       → mk-Quantified-equations1(nms, if sort = pid then id else sort,
6                                  {convert-axiom(eq, lmap, omap, id, pid) | eq ∈ eqs}),
7     mk-Conditional-equation1(eqs, eq)
8       → mk-Conditional-equation1({convert-axiom(a, lmap, omap, id, pid) | a ∈ eqs},
9                                  convert-axiom(eq, lmap, omap, id, pid)),
10    T → axiom)
```

type : *Equation*$_1$ (*Literal-operator-identifier*$_1$ $\overset{m}{\rightarrow}$*Literal-operator-identifier*$_1$) (*Operator-identifier*$_1$ $\overset{m}{\rightarrow}$*Operator-identifier*$_1$) *Sort-identifier*$_1$ *Sort-identifier*$_1$ → *Equation*$_1$

Objective Modify an inherited axioms to include the operators and literals defined by an inheriting sort

Parameters

axiom	The axiom to be modified
lmap	A map from parent literal identifiers into literal identifiers defined for an inheriting sort
omap	A map from parent operator identifiers into operator identifiers defined (implicit or explicit) for an inheriting sort
id	The AS_1 identifier of the inheriting sort
pid	The AS_1 identifier of the parent sort

Result The modified axiom (equation)

Algorithm

Line 2	If the equation is a simple unquantified equation then convert the lefthand side term (*t1*) and the right-hand side term (*t2*).
Line 4-6	If the equation is a quantified equation then replace the sort in the quantification if it is the parent sort and convert the equations contained in the quantified equation.
Line 7-9	If the equation is a conditional equation then convert the restriction (line 8) and convert the restricted equations (line 9).
Line 10	Informal text is left unchanged

$convert\text{-}term(term, lmap, omap) \triangleq$ (3.4.7.15)

```
1  if is-Error-term1(term) then
2    term
3   else
4   (let t = cases term:
5            (mk-Ground-term1(te)     → te,
6             mk-Composite-term1(te) → te) in
7    let t' = cases t:
8            (mk-Identifier1(,)
9               → if is-Ground-term1(term) ∧ t ∈ dom lmap then lmap(t) else t,
10           mk-Conditional-term1(t1, t2, t3)
11              → mk-Conditional-term1(convert-term(t1, lmap, omap),
12                                     convert-term(t2, lmap, omap),
13                                     convert-term(t3, lmap, omap)),
14           T → (let (opid, arglist) = t in
15                let arglist' = ⟨convert-term(arglist[i], lmap, omap) | 1 ≤ i ≤ len arglist⟩ in
16                if opid ∈ dom omap then
17                  (omap(opid), arglist')
18                 else
19                  (opid, arglist'))) in
20   if is-Ground-term1(term) then
21     mk-Ground-term1(t')
22    else
23     mk-Composite-term1(t'))
```

type: $Term_1$ ($Literal\text{-}operator\text{-}identifier_1 \overset{m}{\rightarrow} Literal\text{-}operator\text{-}identifier_1$) ($Operator\text{-}identifier_1 \overset{m}{\rightarrow} Operator\text{-}identifier_1$) $\rightarrow Term_1$

Objective Modify an term to include the operators and literals defined by an inheriting sort

Parameters

term	The term to be modified
lmap	A map from parent literal identifiers into literal identifiers defined for an inheriting sort
omap	A map from parent operator identifiers into operator identifiers defined (implicit or explicit) for an inheriting sort

Result The modified Term

Algorithm

Line 1	If it is the error term then do nothing
Line 4-6	Decompose the term. The replacement takes place regardless of whether it is a ground or a composite term.
Line 7	If the term contains an identifier and the term is a ground term then the identifier denotes a literal and it is replaced if it is in the literal map.
Line 10-13	If the term contains a conditional term then convert the boolean term, the "then" term and the "else" term.
Line 14-19	If the term is an operator application then
Line 14	Decompose the operator application.
Line 15	Convert the argument terms.
Line 16-19	If the operator identifier is in the operator map then fetch the new inherited operator identifier from the map (*omap*) else do not change the operator identifier. Compose the operator application again.
Line 20-23	Compose the term again (reverse of line 4-6)

transform-geninst(*nm*, *instlist*, *adding*)(*dict*) $\triangleq$ (3.4.7.16)

```
 1  (if instlist = ⟨⟩ then
 2     adding
 3    else
 4     (let mk-Geninst0(id, parm) = hd instlist in
 5      let mk-Properties0(lit, op, ax, ma, init) = transform-geninst(nm, tl instlist, adding)(dict) in
 6      let tqual = get-visible-qual(id, GENERATOR)(dict) in
 7      let mk-Id0(, genname) = id in
 8      let mk-GeneratorD(fparm, prop) = dict(tqual) in
 9      if len fparm ≠ len parm then
10        exit("§5.4.1.12.2: Lengths of actual and formal parameter lists in generator must be the same")
11       else
12        (let (tm, litm, opm, constm) = collect-genparms(fparm, parm) in
13         let tm' = tm + [genname ↦ mk-Id0(⟨⟩, nm)] in
14         let mk-Properties0(lit', op', ax', ma', init') = prop in
15         let lit'' = ⟨insert-genparms(lit'[i])(tm', litm, opm, constm) | 1 ≤ i ≤ len lit'⟩,
16             op'' = ⟨insert-genparms(op'[i])(tm', litm, opm, constm) | 1 ≤ i ≤ len op'⟩,
17             ax'' = ⟨insert-genparms(ax'[i])(tm', litm, opm, constm) | 1 ≤ i ≤ len ax'⟩,
18             ma'' = ⟨insert-genparms(ma'[i])(tm', litm, opm, constm) | 1 ≤ i ≤ len ma'⟩ in
19         if init ≠ nil ∧ init' ≠ nil then
20           exit("§5.5.3.3: More than one default assignment")
21          else
22           (let init'' = if init = nil then init' else init in
23            mk-Properties0(lit ⌢ lit'', op ⌢ op'', ax ⌢ ax'', ma ⌢ ma'', init'')))))
```

type: $Name_0$ $Geninst_0$ $Properties_0$ → *Dict* → $Properties_0$

Objective Transform a series of generator instances into AS_0 $Properties_0$.

Parameters

nm	The name of the AS_0 sort definition containing the generator instance.
instlist	The AS_0 generator instance list.
adding	The additional properties specified in the ADDING construct.

Result AS_0 $Properties_0$ to be transformed into AS_1 in the function *transform-partial-typedef*

Algorithm

Line 1	When the generator instance list is empty then return the ADDING properties (the function is recursive).
Line 4	Decompose the next generator instance in the list.
Line 5	Transform the rest of the generator instances and return the *properties* where the properties of the generators in the rest of the list have been added.
Line 6-7	Construct the *Qual* of the generator identifier and decompose the generator identifier.
Line 8	Decompose the generator descriptor.
Line 9-10	The length of the generator actual parameter list must be equal to the length of the generator formal parameter list.
Line 12	Construct four maps which contains the correlation between the formal and actual parameters. *tm* contains the correlation between formal and actual sort parameters, *litm* between formal and actual literal parameters, *opm* between formal and actual operator parameters and *constm* between formal and actual constant parameters.
Line 13	Regard the generator name as a formal parameter and include it in the sort map.

Line 15-18	Replace every formal parameter in the generator body by the corresponding actual parameter i.e. replace the formal parameters in the literal signature (line 15), operator signature (line 16) equations (line 17) and mapping part of the equations (line 18).
Line 19	If this generator instance contains a default assignment then the other generator instances or the ADDING properties must not contain a default assignment.
Line 23	Compose the properties from the properties from this generator instance joined with the properties from the other instances.

insert-genparms(*node*)(*tmap*, *lmap*, *omap*, *constmap*) $\triangleq$ **(3.4.7.17)**

```
 1  cases node:
 2   (mk-Opspec0(nm, sortl, sort)
 3      → mk-Opspec0(insert-parm(nm, omap), ⟨insert-parm(sortl[i], tmap) | 1 ≤ i ≤ len sortl⟩,
 4                   insert-parm(sort, tmap)),
 5    mk-Equation0(t1, t2)
 6      → mk-Equation0(insert-genparms(t1)(tmap, lmap, omap, constmap),
 7                     insert-genparms(t2)(tmap, lmap, omap, constmap)),
 8    mk-Condequation0(eql, eq)
 9      → mk-Condequation0(⟨insert-genparms(eql[i])(tmap, lmap, omap, constmap) | 1 ≤ i ≤ len eql⟩,
10                         insert-genparms(eq)(tmap, lmap, omap, constmap)),
11    mk-Mappingaxiom0(vl, sid, al)
12      → mk-Mappingaxiom0(vl, insert-parm(sid, tmap),
13                         ⟨insert-genparms(al[i])(tmap, lmap, omap, constmap) | 1 ≤ i ≤ len al⟩),
14    mk-Quantequation0(vl, sid, al)
15      → mk-Quantequation0(vl, insert-parm(sid, tmap),
16                          ⟨insert-genparms(al[i])(tmap, lmap, omap, constmap) | 1 ≤ i ≤ len al⟩),
17    mk-Name0(,)
18      → insert-parm(node, lmap),
19    mk-Id0(,)
20      → insert-parm(node, lmap + constmap),
21    mk-Operatorterm0(x, tli)
22      → mk-Operatorterm0(insert-parm(x, omap),
23                         ⟨insert-genparms(tli[i])(tmap, lmap, omap, constmap) | 1 ≤ i ≤ len tli⟩),
24    mk-Condterm0(e1, e2, e3)
25      → mk-Condterm0(insert-genparms(e1)(tmap, lmap, omap, constmap),
26                     insert-genparms(e2)(tmap, lmap, omap, constmap),
27                     insert-genparms(e3)(tmap, lmap, omap, constmap)),
28    mk-Monadterm0(op, e)
29      → mk-Monadterm0(op, insert-genparms(e)(tmap, lmap, omap, constmap)),
30    mk-Infixterm0(e1, op, e2)
31      → mk-Infixterm0(insert-genparms(e1)(tmap, lmap, omap, constmap),
32                      op, insert-genparms(e2)(tmap, lmap, omap, constmap)),
33    mk-Spellingterm0(x)
34      → mk-Spellingterm0(insert-parm(x, lmap)),
35    T → node)
```

type: $(Literal_0 \mid Op_0 \mid Axiom_0 \mid Mappingaxiom_0) \to (Name_0 \xrightarrow{m} Id_0)\ (Name_0 \xrightarrow{m} Literal_0)$
$(Name_0 \xrightarrow{m} (Name_0 \mid Quotedop_0))\ (Name_0 \xrightarrow{m} Term_0) \to (Literal_0 \mid Op_0 \mid Axiom_0 \mid Mappingaxiom_0)$

Objective Replace formal generator parameters by actual generator parameters in the body of a generator

Parameters

node	An object occurring in the body of the generator. The function recursively traverses through the sub-trees of *node*
tmap	The map from formal sort parameters into actual type parameters

lmap	The map from formal literal parameters into actual literal parameters
omap	The map from formal operator parameters into actual operator parameters
constmap	The map from formal constant parameters into actual constant parameters

Result The *node* where the formal parameters have been replaced

Algorithm

Line 2-4	In an operator signature, consider the operator name (*nm*), the argument sort list (*sortl*) and the result sort (*sort*).
Line 5-7	In an equation, consider the two terms *t1* and *t2*.
Line 8-10	In a conditional equation, consider the restriction (*t1*) and the restricted equations (*t2*).
Line 11-13	In a mapping equation, consider the sort identifier (*sid*) and the contained equations (*al*).
Line 14-16	In a quantified equation, consider the sort identifier (*sid*) and the contained equations (*al*).
Line 17	If the node is a $Literal_0$ $Name_0$, consider it using the literal map.
Line 19	If the node is an identifier (occurring in an equation) then consider it using the literal map and the constant map (literal formal parameters may also be used in equations).
Line 21-23	In an operator application, consider the operator identifier (*x*) and the argument terms (*tli*).
Line 24-27	In a conditional term, consider the boolean term (*e1*) the "then" term (*e2*) and the "else" term (*e3*)
Line 28-32	in a monadic term or a dyadic term, consider the contained term(s).
Line 33	In a spelling term, consider the contained identifier.
Line 35	Other kinds of *nodes* cannot or must not contain any formal parameters

insert-parm(x, map) $\triangleq$ (3.4.7.18)

```
 1  cases x:
 2   (mk-Id0(q, nm)
 3      → if nm ∈ dom map then
 4           if q = ⟨⟩ then
 5              cases map(nm):
 6               (mk-Name0(,)
 7                  → mk-Id0(⟨⟩, map(nm)),
 8                mk-Quotedop0()
 9                  → mk-Qualop0(⟨⟩, map(nm)),
10                mk-Nmclass0()
11                  → exit("§5.4.1.12: Name class cannot be used in equations"),
12                mk-String0()
13                  → mk-Stringterm0(⟨⟩, map(nm)),
14                T → map(nm))
15             else
16              exit("§5.4.1.12: Generator formal name is qualified")
17          else
18           x,
19    mk-Name0(,)
20      → if x ∈ dom map then
21           cases map(x):
22            (mk-Name0(,),
23             mk-Nmclass0(),
24             mk-String0()
25               → map(x),
26             mk-Quotedop0()
27               → map(x),
28             T → exit("§5.4.1.12: Generator constant parameter used in literal signature"))
29          else
30           x,
31    T → x)
```

type: $(Id_0 \mid Name_0 \mid Quotedop_0)\ (Name_0 \xrightarrow{m} (Term_0 \mid Literal_0 \mid Quotedop_0)) \rightarrow (Term_0 \mid Literal_0 \mid Quotedop_0)$

Objective

Test whether an identifier or a name is a formal generator parameter and if so, replace it by the corresponding actual parameter

Parameters

x — The identifier or name to be tested upon. In order to simplify the applying function *insert-genparms*, *x* may also be an $Quotedop_0$ in which case *x* is returned unchanged.

map — A map containing the association between formal parameters and actual parameters for a certain parameter class depending on the context in which *insert-parm* is applied. The class is either type formal parameters, operator formal parameters or the composition of literal and constant formal parameters.

Result

The corresponding actual parameter if *x* is a formal parameter, otherwise *x*

Algorithm

Line 2 — If the object in hand is an identifier and the name part it is in the map (*map*) then the identifier denotes a formal parameter in which case it must not be qualified (line 4 and line 16).

Line 6 — If the actual parameter is a name (i.e. a literal or operator name) then return it enclosed in an identifier (because it is used in applying context).

Line 8	If the actual parameter is a quoted operator then return it enclosed in an operator identifier ($Qualop_0$)
Line 10	A nameclass cannot be used in applying context.
Line 12	If the actual parameter is a character string then return it enclosed in a string literal identifier ($Stringterm_0$).
Line 14	Otherwise the actual parameter is a term in which case the term is return.
Line 18	If the identifier is not a formal parameter then return the identifier unchanged.
Line 19	If the object in hand is a name and it is in the map then it is a formal parameter used in defining context.
Line 22-24	If the actual parameter is a name or a name class or a character string then it is returned.
Line 26	If the actual parameter is a quoted infix operator then the contained infix operator is returned.
Line 28	In other cases, the actual parameter denotes a term which is not an allowed parameter in defining context.
Line 30	If the name is not a formal parameter then return the name unchanged

collect-genparms(*fparm*, *parm*) $\triangleq$ (3.4.7.19)

```
1  (if fparm = ⟨⟩ then
2     ([], [], [], [])
3    else
4     (let (trest, litrest, oprest, constrest) = collect-genparms(tl fparm, tl parm) in
5      let exclamation =
6          if is-Id0(hd parm) then
7            (let mk-Name0(, exc) = s-Name0(hd parm) in
8             exc ≠ nil)
9           else
10           false in
11       cases hd fparm:
12       (mk-Sortparm0(nm)
13          → if is-Id0(hd parm) ∧ ¬exclamation then
14               (trest + [nm ↦ hd parm], litrest, oprest, constrest)
15             else
16               exit("§5.4.1.12.2: TYPE actual generator parameter must be a identifier"),
17        mk-Opparm0(nm)
18          → if (is-Id0(hd parm) ∧ s-Qualifier0(hd parm) = ⟨⟩) then
19               (trest, litrest, oprest + [nm ↦ s-Name0(hd parm)], constrest)
20             else
21               if is-Quotedop0(hd parm) then
22                 (trest, litrest, oprest + [nm ↦ hd parm], constrest)
23               else
24                 exit("§5.4.1.12.2: OPERATOR actual generator parameter must be an operator signature"),
25        mk-Litparm0(nm)
26          → cases hd parm:
27              (mk-String0(str)
28                 → (trest, litrest + [nm ↦ str], oprest, constrest),
29               mk-Id0(q, nam)
30                 → if q = ⟨⟩ ∧ ¬exclamation then
31                      (trest, litrest + [nm ↦ nam], oprest, constrest)
32                    else
33                      exit("§5.4.1.12.2: LITERAL actual generator parameter must be a literal signature"),
34               mk-Nmclass0()
35                 → (trest, litrest + [nm ↦ hd parm], oprest, constrest),
36               T → exit("§5.4.1.12.2: LITERAL actual generator parameter must be a literal signature")),
37        mk-Termparm0(nm)
38          → if is-Nmclass0(hd parm) ∨ is-Quotedop0(hd parm) ∨ exclamation then
39               exit("§5.4.1.12.2: CONSTANT actual generator parameter must be a term")
40             else
41               (trest, litrest, oprest, constrest + [nm ↦ hd parm]))))
```

type: $Genparm_0^*$ $Genactparm_0 \rightarrow (Name_0 \overset{m}{\rightarrow} Id_0)$ $(Name_0 \overset{m}{\rightarrow} Literal_0)$ $(Name_0 \overset{m}{\rightarrow} (Name_0 \mid Quotedop_0))$ $(Name_0 \overset{m}{\rightarrow} Term_0)$

Objective

Construct four maps which associate the four kinds of formal parameters to the corresponding actual parameters for a generator instance. Prior to the application of this function, it is checked that the two lists have the same length

Parameters

fparm	The list of generator formal parameters
parm	The list of generator actual parameters

Algorithm

Line 1	When through the formal parameter list return the empty maps (the function is recursive).

Line 4	Construct the maps for the rest (all but the first) of the formal parameters.
Line 5-10	Let *exclamation* be **true** if the actual parameter is an Id_0 where the $Name_0$ has an exclamationmark.
Line 12-16	If the formal parameter is a sort parameter then the actual parameter must be an identifier without any exclamationmark. The association between the formal name and the identifier is added to the sort map.
Line 17-24	If the formal parameter is an operator parameter then the actual parameter must be an unqualified identifier (i.e. a name) or a quoted infix operator
Line 19 and *Line 22*	The association between the formal name and the actual name respectively actual infix operator is added to the operator map.
Line 25	If the formal parameter is a literal parameter then the actual parameter must be an unqualified identifier or an unqualified string or a name class.
Line 27 and *Line 34*	The association between the formal name and the actual name or string or nameclass is added to the literal map.
Line 37-41	If the formal parameter is a constant parameter then the actual parameter must be a term (i.e. not a name class or a quoted infix operator) and if the term is an Id_0 then it must not contain an exclamationmark.
Line 41	The association between the formal name and the term is added to the constant map

transform-syntype(mk-*Syntypedef*$_0$(*nm*, *pid*, *initial*, *vallist*, *tnm*))(*dict*) $\triangleq$ (3.4.7.20)

```
 1  (let tq = get-visible-qual(pid, TYPE)(dict) in
 2   if is-recursive-sort(tq, {})(dict) then
 3     exit("§5.4.1.9: Syntype is based on itself")
 4    else
 5     if tnm ∉ {nm, nil} then
 6       exit("§5.4.1.9: Ending name in syntype definition is different from defining name")
 7      else
 8       (let tqual = get-parent(tq)(dict) in
 9        let initial_1 = cases dict(tq):
10                         (mk-SortD(, , init, )
11                            → init,
12                          mk-SyntypeD(, , init, )
13                            → init) in
14        let expr_1 = if initial = nil then
15                       initial_1
16                     else
17                      (let (as_1 tree, , ) =
18                           transform-expr(initial, CONSTANT, {tqual})(dict) in
19                       as_1 tree) in
20        let (, as_1 valset) = transform-valueset({tqual}, vallist)(dict) in
21        let as_1 id = make-as_1-identifier(tqual)(dict) in
22        let nm' = if is-ServiceD(dict(dict(SCOPEUNIT))) then
23                    create-unique-name()
24                  else
25                   nm in
26        let as_1 dcl = mk-Syn-type-definition_1(name-to-name_1(nm'), as_1 id, as_1 valset) in
27        let synqual = dict(SCOPEUNIT) ⌢ ⟨(TYPE, nm')⟩ in
28        let d = [dict(SCOPEUNIT) ⌢ ⟨(TYPE, nm)⟩ ↦ mk-SyntypeD(tq, synqual, expr_1, as_1 valset)] in
29        ({as_1 dcl}, d)))
```

type : *Syntypedef*$_0$ → *Dict* → *Syn-type-definition*$_1$-**set** *Dict*

Objective Transform a syntype definition into AS_1

Parameters A syntype definition containing:

nm	The name of the syntype.
pid	The parent sort identifier.
initial	The initial value for variables of the syntype.
vallist	The allowed value set for variables of the sort.
tnm	The name ending the definition

Result See *transform-decllist*

Algorithm

Line 1	Extract the *Qual* for the specified parent sort.
Line 2	The parent sort must not be based on this syntype
Line 5	The tailing name must be equal to the syntype name is specified
Line 8	Extract the partial type of the specified parent sort.
Line 9	Let $initial_1$ denote the initial expression of *pid*.
Line 14-19	Construct the AS_1 version of the initial expression. If it is omitted then the initial expression of *pid* is used.
Line 20	Transform the value set
Line 22	If the syntype definition occurs in a service then create a new name for the syntype
Line 27	Construct the *Qual* to be used when deriving the AS_1 identifier of the syntype (at the places where the syntype is used).
Line 28-29	Return the AS_1 syntype definition ($as_1\,dcl$) and the *Dict* contribution containing the syntype descriptor (d).

transform-valueset(*tqualset*, *valset*)(*dict*) $\triangleq$ (3.4.7.21)

```
1  (if valset = ⟨⟩ then
2     (let boolq = get-predef-sort("BOOLEAN")(dict) in
3      let orq = boolq ⌢ ⟨(OPERATOR, (OR, ⟨boolq, boolq⟩, boolq))⟩ in
4      (tqualset, mk-Range-condition1(make-as1-identifier(orq)(dict), {})))
5    else
6     (let mk-Condition0(cr, expr) = hd valset in
7      let (tset, as1 range) = transform-valuerange(tqualset, cr, expr)(dict) in
8      let (trest, mk-Range-condition1(orid, condset)) = transform-valueset(tset, tl valset)(dict) in
9      (trest, mk-Range-condition1(orid, {as1 range} ∪ condset))))
```

type: *Sortqual*-set $Condition_0^*$ → *Dict* → *Sortqual*-set *Range-condition*$_1$

Objective Transform a value set into AS_1.

Parameters

tqualset	A set of possible (legal) sorts.
valset	The list of value ranges. When the value set occurs in a syntype definition the *tqualset* contains the parent of the syntype only. When the value set occurs in a decision answer the *tqualset* contains some subset (also depending of other answers) of the possible sorts for the question expression.

Result The set of possible (legal) sorts after having dealt with the value set, i.e. the resulting set is a subset of *tqualset*. Also the AS_1 value set is returned ($Range\text{-}Expression_1^*$).

Algorithm

Line 1-4	When through all the elements in the value set then return the set of possible sorts and an empty *Range-condition*$_1$ containing the AS$_1$ identifier of the boolean OR operator.
Line 6	Take the first element in the set (list) which consists of an optional operator or an optional expression (*cr*) followed by an expression (*expr*).
Line 7-9	Transform a value range and use the resulting (possibly restricted) sort set (*tset*) during the transformation of the rest of the value set.

transform-valuerange(*tqualset*, *cr*, *expr*)(*dict*) $\triangleq$ (3.4.7.22)

```
 1  (let bq = get-predef-sort("BOOLEAN")(dict) in
 2   let cr' = if cr = nil then EQ else cr in
 3   let isrel = cr' ∈ {NE, EQ, GT, LT, LE, GE} in
 4   let cr'' = if isrel then cr' else LE in
 5   let tqualset' =
 6       if isrel then
 7         tqualset
 8        else
 9         (let (,tqset,) = transform-expr(cr, CONSTANT, {bq})(dict) in
10          tqset ∩ tqualset) in
11   let opqset =
12       {opqual ∈ all-visible-operators(⟨⟩, cr'', dict(SCOPEUNIT))(dict) |
13        is-OperatorD(dict(opqual)) ∧
14        (let (, (,argl, res)) = opqual[len opqual] in
15         len argl = 2 ∧
16         res = bq ∧ argl[1] = argl[2] ∧ elems argl ⊂ tqualset')} in
17   if opqset = {} then
18     exit("§5.4.1.9.1: Ordering operator is not defined for the sort of the range condition")
19    else
20     (let tset = tqualset' ∩ {sq ∈ dom dict | is-SortD(dict(sq)) ∧
21                                              (∃q ∈ opqset)((OPERATOR, (cr'', ⟨sq, sq⟩, bq)) = q[len q])} in
22      if card tset ≥ 1 then
23        (tset, nil)
24       else
25        (let (as1tree,,) = transform-expr(expr, CONSTANT, tset)(dict) in
26         let sort ∈ tset in
27         let opqual = sort ⌢ ⟨(OPERATOR, (cr'', ⟨sort, sort⟩, bq))⟩ in
28         if isrel then
29           (tset, mk-Open-range1(make-as1-identifier(opqual)(dict), as1tree))
30          else
31           (let (as1tree',,) = transform-expr(cr, CONSTANT, tset)(dict) in
32            let r1 = mk-Open-range1(make-as1-identifier(opqual)(dict), as1tree'),
33                r2 = mk-Open-range1(make-as1-identifier(opqual)(dict), as1tree) in
34            let andq = bq ⌢ ⟨(OPERATOR, (AND, ⟨sort, sort⟩, {bq}))⟩ in
35            (tset, mk-Closed-range1(make-as1-identifier(andq)(dict), r1, r2))))))
```

type : *Sortqual*-**set** [*Expr*$_0$ | *Relop*$_0$] *Expr*$_0$ $\rightarrow$ *Dict* $\rightarrow$ *Sortqual*-**set** [*Condition-item*$_1$]

Objective Transform an AS$_0$ *Condition*$_0$ into an AS$_1$ *Condition-item*$_1$

Parameters

tqualset	The set of legal sorts prior to transformation of the value range
cr	The relational operator or first expression occurring in the value range
expr	The (second) expression of the value range

Result The set of legal sorts after the transformation of the value range and the constructed AS_1 *Condition*$_1$. The sort set is used in the transformation of decision actions where this function is applied twice. First time in order to derive the sort and the second time in order to transform the value range

Algorithm

Line 1	Construct the *Qual* of the boolean sort.
Line 2	If neither a relational operator nor an expression is specified in the value range then the equality operator applies.
Line 4	If *cr* is an expression then the less-equal operator (LE) is used as relational operator.
Line 5-10	If *cr* is an expression then the set of legal sorts (*tqualset*) is restricted by the legal sorts for that expression.
Line 11-16	Construct the *Qual* set of legal relational operators such that the length of the argument list is equal to 2 and the result sort is boolean and the two arguments are of the same sort and that sort is one of the legal sorts in the context where the value range is used (if there is more than one element in the set then the operator name (*cr*) is overloaded).
Line 17-18	If no such operators exits then it is an error.
Line 20-21	Construct a new set of legal sorts by restricting the old set of legal sorts (*tqualset'*) to contain only those sorts which can be used on the operators in *opqset*.
Line 22-23	If there is more than one element in the restricted set of sorts then the value range is (still) ambiguous and no value range is returned (**nil** is returned).
Line 25	Evaluate the expression again. This time, *tset* contains exactly one sort and therefore an AS_1 expression is returned (if it is otherwise well-formed).
Line 26	Denote the sort in *tset* by *sort*.
Line 27	Construct the *Qual* of the operator used.
Line 28-29	If *cr* denotes a relational operator then return the set containing the sort and the *Condition-item*$_1$ which is an *Open-range*$_1$.
Line 31	If *cr* is an expression then transform the expression (again).
Line 32-33	Construct the two *Open-range*$_1$s which together with the AND operator constitutes the *Closed-range*$_1$.
Line 34	Construct the *qual* of the AND operator and
Line 35	Return the resulting set of sorts (containing one element) and the *Closed-range*$_1$ containing the AS_1 identifier of the AND operator and the two *Closed-range*$_1$s.

transform-synonymdef(mk-*Synonymdef*$_0$(*nm*, *tid*, *expr*))(*dict*) $\triangleq$ **(3.4.7.23)**

```
 1  (let squal = dict(SCOPEUNIT) ⌢ ⟨(VALUE, nm)⟩ in
 2   let sortset = if tid = nil then
 3                    all-visible-sorts(dict)
 4                 else
 5                    {get-parent(get-visible-qual(tid, TYPE)(dict))(dict)} in
 6   let (,tpset,) = transform-expr(expr, CONSTANT, sortset)(dict + [squal ↦ mk-ErrorD()]) in
 7   if card tpset = 1 then
 8     (let tp ∈ Qual be s.t. tp ∈ tpset in
 9      ([squal ↦ mk-SynD(tp, expr)]))
10    else
11     exit("§5.4.1.13: Sort of synonym cannot be uniquely determined"))
```

type: *Synonymdef*$_0$ → *Dict* → *Dict*

Objective Transform a synonym definition into AS_1

Parameters An AS_0 synonym definition containing

nm	The synonym name
tid	The optional sort identifier
expr	The ground expression

Result See *transform-decllist*

Algorithm

Line 1	Construct the *Qual* denoting the synonym.
Line 2-5	The sort of the synonym must be found in the set of sorts which are visible to the synonym definition if the sort identifier (*tid*) is absent ,otherwise the sort of the synonym is *tid*.
Line 6	Transform the expression contained in the synonym definition and return the subset of sorts (*tpset*) which matches both the synonym sort. The resulting AS_1 expression is not used. As the synonym identifier must not be used in *expr squal* denotes a *ErrorD* during evaluation of the expression.
Line 7	Only one sort is allowed to match the sort of the synonym and the sort of the expression.
Line 8-9	Construct the *Dict* contribution containing the synonym descriptor.

3.4.7.1 Partial type Definitions

transform-sortdef$(nm, prop, parent)(dict) \triangleq$ (3.4.7.1.1)

```
 1  (let nm' = if is-ServiceD(dict(dict(SCOPEUNIT))) then create-unique-name() else nm in
 2   let newqual = dict(SCOPEUNIT) ↷ ⟨(TYPE, nm')⟩ in
 3   let descr = dict(SCOPEUNIT) ↷ ⟨(TYPE, nm)⟩ in
 4   let dict' = dict + [SCOPEUNIT ↦ descr] in
 5   let prop' = transform-nameclass(prop) in
 6   let prop'' = add-ordering(prop', descr)(dict) in
 7   let mk-Properties0(lit, oplist, axioms, mapping, term) = prop'' in
 8   let (eqop, eqax) = add-equality(descr)(dict) in
 9   let (expr1, , ) = transform-expr(term, CONSTANT, {descr})(dict) in
10   if card elems lit ≠ len lit then
11     exit("§5.2.2: Literal defined twice in a partial type definition")
12    else
13     (let litd = [descr ↷ ⟨(LITERAL, nm)⟩ ↦ mk-LiteralD(descr) | nm ∈ elems lit],
14          (as1 op, opd) = transform-typing(oplist ↷ eqop)(dict'),
15          as1 ax = transform-axioms(axioms ↷ eqax, AXIOMS)(dict'),
16          as1 mapping = transform-axioms(mapping, MAPPING)(dict') in
17      let sortid = make-as1-identifier(newqual)(dict) in
18      let as1 litset = {mk-Literal-signature1(nm, sortid) | nm ∈ card lit} in
19      let concaxioms1 = make-as1-concaxioms(dom litd)(dict) in
20      let typedef = dict(DATATYPEDEF) in
21      let mk-Data-type-definition1(typename, union, sorts, sigs, eqs ∪ concaxioms1) = typedef in
22      let datatypedef' = mk-Data-type-definition1(typename, union, sorts ∪ {name-to-name1(nm')},
23                                                  sigs ∪ as1 op ∪ as1 litset,
24                                                  eqs ∪ as1 ax ∪ as1 mapping ∪ concaxioms1) in
25      let sortdescr = mk-SortD(as1 ax ∪ as1 mapping, parent, expr1, newqual) in
26      (litd + opd + [descr          ↦ sortdescr,
27                     DATATYPEDEF ↦ datatypedef']))))
```

type : $Name_0\ Properties_0\ [Sortqual] \rightarrow Dict \rightarrow Dict$

Objective

Transform the *Properties*$_0$ of a sort definition and add the resulting AS$_1$ properties to the *Data-type-definition*$_1$ contained in the *DATATYPE-DEF* entry in the *Dict* and also add the descriptors for the literals, for the operators and for the sort to *Dict*

Parameters

nm	The AS$_0$ name of the sort being defined
prop	The AS$_0$ properties to be transformed
parent	The *Qual* of the parent sort in the case where the properties originated from the inheriting sort i.e. *parent* denotes the identifier of the sort from which this sort inherits. It is put in the descriptor for the sort descriptor *SortD* (see the definition of *SortD*).

Result

The *Dict* updated with the AS$_1$ properties and updated with the various descriptors

Algorithm

Line 1-2	Construct a *Qual* to be used when constructing the AS$_1$ name. If the sort is defined in a service a new unique name is used in the *Qual*
Line 3-4	Update SCOPEUNIT indicating that the properties are evaluated in the context of the sort *nm* and update *Dict* to include information of the new scopeunit.
Line 5	Modify the properties such that all the name classes in the literal definition are expanded to a sequence of literals.
Line 6	Modify the properties such that any ORDERING in the operator signatures are replaced by the operator signatures and equations reflecting the ordering properties.
Line 7	Decompose the *Properties*$_0$.
Line 8	Construct the AS$_0$ operators and equations reflecting the equality properties.
Line 9	Transform the DEFAULT expression in the sort definition into AS$_1$
Line 10	Number of distinct literal names in the literal signature must be equal to the length of the literal signature list.
Line 13	Construct the descriptors for all the literals in the literal signature.
Line 14	Construct the AS$_1$ *Operator-signature*$_1$s (*as*$_1$ *op*) and operator descriptors for the operators in the operator signatures and for the equality operators.
Line 15	Construct the AS$_1$ equations from the AS$_0$ equations (axioms) which are the equations specified in the properties joined with the equality equations.
Line 17	Construct the equations corresponding to the MAP part of the AS$_0$ properties.
Line 18	Construct the AS$_1$ literal signatures.
Line 19	Construct the AS$_1$ equations implied from any defined character string literal (the implicit concatenation equations).
Line 21	Extract and decompose the current *Data-type-definition*$_1$.
Line 22-23	Construct a new *Data-type-definition* which is the old one updated to include the constructed AS$_1$ properties.
Line 25	Construct the descriptor for the sort.
Line 26	Return the *Dict* containing the descriptors for the literals, the descriptors for the operators, the descriptor for the sort and the modified *Data-type-definition*$_1$

transform-nameclass(**mk-***Properties*$_0$(*litlist*, *o*, *a*, *m*, *t*)) $\triangleq$ (3.4.7.1.2)

```
1  if (∃lit ∈ elems litlist)(is-Nmclass0(lit)) then
2    (let i ∈ ind litlist be s.t. is-Nmclass0(litlist[i]) in
3     let mk-Nmclass0(regexpr) = litlist[i] in
4     let stringset = {str ∈ Char+ | is-in-regular-expr(str, regexpr)} in
5     let nameset = form-names-and-strings(stringset) in
6     let namelist be s.t. card elems namelist = card nameset ∧
7         len namelist = card nameset ∧
8         (∀i1, i2 ∈ ind namelist)
9          (namelist[i1] < namelist[i2] ⊃ i1 < i2) in
10    let litlist' = ⟨litlist[n] | 1 ≤ n < i⟩ ⌢ namelist ⌢ ⟨litlist[n] | i < n ≤ len litlist⟩ in
11    transform-nameclass(mk-Properties0(litlist', o, a, m, t)))
12   else
13    mk-Properties0(litlist, o, a, m, t)
```

type : *Properties*$_0$ $\rightarrow$ *Properties*$_0$

Objective Modify some AS$_0$ properties such that any contained name classes are expanded into a sequence of literal names

Parameters The AS$_0$ properties containing

litlist	The literal signature list
o	The operator signature list
a	The equations
m	The mapping equations
t	The term denoting the DEFAULT value

Result The modified AS$_0$ properties

Algorithm

Line 1	If there (still) exist a name class in the literal signatures then
Line 2-3	Let *i* denote the index to the element in the literal signatures list which is a name class and decompose the name class (line 3).
Line 4	Construct the set of (Meta-IV) character strings which satisfy the regular expression denoted by *regexpr*.
Line 5	From this set of character strings, construct a *nameset* consisting of AS$_0$ names and character strings.
Line 6-9	Transform the set into an alphabetic ordered list.
Line 10	Replace the name class in the literal signatures by the constructed list and
Line 11	Replace any further name classes.
Line 13	If no name classes are left then return the properties

form-names-and-strings(*stringset*) $\triangleq$ (3.4.7.1.3)

```
1  if stringset = {} then
2    {}
3   else
4    (let string ∈ stringset in
5     if len string ≥ 2 ∧ string[1] = "'" ∧ string[len string] = "'" then
6       {mk-String0(⟨string[i] | 2 ≤ i ≤ len string − 1⟩)} ∪
7       form-names-and-strings(stringset \ {string})
8      else
9       (let string' = check-name-syntax(string) in
10       {mk-Name0(string', nil)} ∪
11       form-names-and-strings(stringset \ {string})))
```

type : *Char****-set** $\rightarrow$ (*String*$_0$ | *Name*$_0$)**-set**

Objective From a set of Meta-IV character strings form a set of AS_0 names and character strings

Algorithm

Line 1	When through, return nothing (the function is recursive).
Line 4	Take an element of the set and denote it by *string*.
Line 5	*string* denotes a AS_0 character string if the first character and the last character are single quotes.
Line 6-7	Return the AS_0 character string together with the rest of the names and string.
Line 9-11	If it is not a character string it denotes an AS_0 name.
Line 9	Remove/replace space characters in accordance with the semantics of the underline character, convert the name to uppercase and check the spelling against the lexical rules defined in Z.100.
Line 10-11	Return the AS_0 name together with the rest of the names and strings

is-in-regular-expr(*string*, *regexpr*) $\triangleq$ (3.4.7.1.4)

```
1  ( cases regexpr:
2    (mk-Orregexp0(reg, partreg)
3       → is-in-regular-expr(string, reg) ∧
4         is-in-regular-expr(string, partreg),
5     mk-Andregexp0(reg, partreg)
6       → (∃(str1, str2))(str1 ⌢ str2 = string ∧
7                         is-in-regular-expr(str1, reg) ∧
8                         is-in-regular-expr(str2, partreg)),
9     mk-Rngregexp0(str1, str2, exptp)
10      → is-in-regular-range(string, str1, str2, exptp),
11    mk-Singregexp0(str1, exptp)
12      → is-in-regular-single(string, str1, exptp),
13    mk-Parenregexp0(regexp, exptp)
14      → is-in-regular-par(string, regexp, exptp)))
```

type: *Char** *Regularexp*$_0$ → *Bool*

Objective Test whether a Meta-IV character string is included in a regular expression

Parameters

string	The Meta-IV character string
regexpr	The regular expression

Result True if success

Algorithm

Line 2-4	If the regular expression consist of two OR'd regular expressions then the result is true if the string is defined in one of them.
Line 5-8	If the regular expression consist of two concatenated regular expressions then the result is true if there exist two strings such that they form *string* when they are concatenated and such that the first sub-string is defined in the first regular expression and the second sub-string is defined in the second sub-string.
Line 9	See *is-in-regular-range*.
Line 11	See *is-in-regular-single*.
Line 13	See *is-in-regular-par*

is-in-regular-range(*string*, mk-*String*$_0$(*str*1), mk-*String*$_0$(*str*2), *exptp*) $\triangleq$ (3.4.7.1.5)

```
 1  (if len str1 ≠ 1 ∨ len str2 ≠ 1 then
 2     exit("§5.4.1.14: Length of character strings in regular interval is not equal to one")
 3    else
 4     (if len string = 0 ∧ exptp = MULT then
 5        true
 6       else
 7        (len string ≤ 1 ∧
 8         (∀ch ∈ elems string)
 9          ((hd str1 ≤ ch ∧
10            ch ≤ hd str2 ∧
11            ( cases exptp:
12              (mk-Name0(numstr, )
13                 → if elems numstr ⊂ {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"} then
14                      (let i = form-integer(numstr) in
15                       len string = i)
16                     else
17                      exit("§5.4.1.14: Repetition name in name class must denote a natural number"),
18               T → true)))))))
```

type : *Char** *String*$_0$ *String*$_0$ [*Regexpexp*$_0$] → *Bool*

Objective Test whether a string is defined in a regular expression which is a character range possibly repeated a number of times

Parameters

string	The string to be tested upon
str1,str2	The character sequences denoting the lower and upper bounds
exptp	The optional repetition factor

Result True if success

Algorithm

Line 1-2	The length of the lower and upper bounds must be equal to one.
Line 4	The empty string can only be in the range if the repetition factor is specified and it allows for empty strings.
Line 7-10	Otherwise the length must be greater or equal to one and for all characters (*ch*) in the string it must hold that their representation values are greater or equal to the lower bound (line 8) and that the representation values are less than or equal to the upper bound.
Line 11-17	If the repetition factor is a name then it must also hold that it denotes a natural literal name (it must consist of digits) and that the value it denotes is equal to the length of the string

form-integer(*str*) $\triangleq$ (3.4.7.1.6)

```
 1  (let ch = str[len str] in
 2   let n = cases ch:
 3             ("0" → 0,
 4              "1" → 1,
 5              "2" → 2,
 6              "3" → 3,
 7              "4" → 4,
 8              "5" → 5,
 9              "6" → 6,
10              "7" → 7,
11              "8" → 8,
```

```
12              "9" → 9) in
13    if tl str = ⟨⟩ then n else form-integer(⟨str[i] | 1 ≤ i ≤ len str⟩) * 10 + n)
```

type: $Char^+ \rightarrow Intg$

Objective Convert a sequence of characters (a spelling) to a Meta-IV *Intg* value

Algorithm

Line 1	Extract the last character in the character list.
Line 2-12	Convert the character to a Meta-IV *Intg* value.
Line 13	If it is the only character in the list then return the value else multiply the value obtained from the rest of the list by 10 and add the value obtained from the character to the result

is-in-regular-single(*string*, mk-$String_0$(*str*), *exptp*) $\triangleq$ (3.4.7.1.7)

```
 1  ( cases exptp:
 2    (MULT
 3       → (∃n)(conc ⟨str | 0 ≤ i ≤ n⟩ = string),
 4     PLUS
 5       → (∃n)(conc ⟨str | 1 ≤ i ≤ n⟩ = string),
 6     mk-Name0(numstr, )
 7       → if elems numstr ⊂ {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"} then
 8            (let n = form-integer(numstr) in
 9             conc ⟨str | 1 ≤ i ≤ len n⟩ = string)
10           else
11            exit("§5.4.1.14: Repetition name in name class must denote a natural number"),
12     nil
13       → str = string))
```

type: $Char^*$ $String_0$ [$Regexpexp_0$] $\rightarrow$ *Bool*

Objective Test whether a certain string can be formed by repeating another string a number of times

Parameters

string	The string to be formed
str	The string to be repeated
exptp	The repetition factor

Result True if success

Algorithm

Line 2	If the repetition factor is an asterisk then there must exist a value *n* such that repetition (concatenation) of *str n* times forms *string.*
Line 4	If the repetition factor is a plus then there must exist a non-zero value *n* such that repetition (concatenation) of *str n* times forms *string.*
Line 6	If the repetition factor is a name it must denote a natural number and the result is true if *string* can be formed by repeating *str* that number of times.
Line 12	If the repetition factor is omitted, the strings must be equal

is-in-regular-par(*string*, *regexp*, *exptp*) $\triangleq$ **(3.4.7.1.8)**

```
 1  cases exptp:
 2   (MULT
 3      → (∃(str, n))(conc ⟨str | 0 ≤ i ≤ n⟩ = string ∧
 4                    is-in-regular-expr(str, regexp)),
 5    PLUS
 6      → (∃(str, n))(conc ⟨str | 1 ≤ i ≤ n⟩ = string ∧
 7                    is-in-regular-expr(str, regexp)),
 8    mk-Name0(numstr, )
 9      → if elems numstr ⊂ {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"} then
10          (let n = form-integer(numstr) in
11           (∃str)(conc ⟨str | 1 ≤ i ≤ n⟩ = string ∧
12                  is-in-regular-expr(str, regexp)))
13         else
14          exit("§5.4.1.14: Repetition name in name class must denote a natural number"),
15    nil
16      → is-in-regular-expr(string, regexp))
```

type : *Char** *Regularexp*$_0$ [*Regexpexp*$_0$] → *Bool*

Objective Test whether a certain string can be formed by repeating a sub-string denoted by a regular expression a number of times

Parameters

string	The string to be tested upon
regexp	The regular expression which is to be repeated
exptp	The optional repetition factor

Result True if success

Algorithm

Line 2-4	If the repetition factor is an asterisk then there must exist a sub-string *str* which is defined by the regular expression *regexp* and a number *n* such that repetition of the sub-string *n* times forms the string
Line 5-7	If the repetition factor is a plus then there must exist a sub-string *str* which is defined by the regular expression *regexp* and a non-zero number *n* such that repetition of the sub-string *n* times forms the string.
Line 8-14	If the repetition factor is a name it must denote a natural number and the result is true if there exist a sub-string *str* defined by the regular expression *regexp* such that repetition of the sub-string the specified number of times forms *string*.
Line 15	If the repetition factor is omitted, the result is true if the string is defined by the regular expression

transform-axioms(*axioms*, *context*)(*dict*) $\triangleq$ **(3.4.7.1.9)**

```
1  (if axioms = ⟨⟩ then {} else (transform-axiom(hd axioms, context)(dict) ∪
2                                transform-axioms(tl axioms, context)(dict)))
```

type : *Axiom*$_0$* *Context* → *Dict* → *Equations*$_1$

Objective Transform a list of AS_0 equations (axioms) into a set of AS_1 equations

Parameters

axioms	The axioms to be transformed

context — The context is which the axioms are transformed. The context either denotes the normal axioms (AXIOMS) or the mapping axioms (MAPPING)

Result The generated AS_1 equations

Algorithm

Line 1 — When through, return the empty set.

Line 1-2 — Return the set of equations consisting of the set corresponding to the first equation (line 1) unified with the rest of the equations (line 2)

transform-axiom(*axiom*, *context*)(*dict*) $\triangleq$ (3.4.7.1.10)

```
1  ( cases axiom:
2    (mk-Equation0(,)
3       → {transform-equation(axiom, context)(dict)},
4     mk-Condequation0(eql, eq)
5       → (let eql' = ⟨insert-equals-true(eql[i])(dict) | i ≤ i ≤ len eql⟩,
6              eq' = insert-equals-true(eq)(dict) in
7          {transform-condequation(mk-Condequation0(eql', eq'), context)(dict)}),
8     mk-Quantequation0(,,)
9       → {transform-quantequation(axiom, context)(dict)},
10    mk-Mappingaxiom0(,,)
11      → transform-mapping(axiom)(dict),
12    T → if is-Stringterm0(axiom) ∧ s-Qualifier0(axiom) = ⟨⟩ then
13          {mk-Informal-text1(axiom)}
14        else
15          (let axiom' = insert-equals-true(axiom)(dict) in
16           {transform-equation(axiom', context)(dict)})))
```

type: $Axiom_0$ *Context* → *Dict* → $Equations_1$

Objective Transform an AS_0 axiom into a set of AS_1 equations. Except for in the case of a mapping axiom, the set consist of only one element

Parameters

axiom — The axiom to be transformed

context — The context is which the transformation takes place (see *transform-axioms*)

Result The generated AS_1 equations

Algorithm

Line 2 — If the axiom is a simple equation then transform the equation.

Line 4-7 — If the axiom is a conditional equation then replace boolean terms in the restriction (*eql*) and in the restricted equation (*eq*) by a simple equation and transform the resulting conditional equation.

Line 8-10 — If the axiom is a quantified equations (line 8) or a mapping equation (line 10) then transform it.

Line 12-13 — Otherwise the axiom is a term which denotes informal text it the term is an unqualified character string.

Line 15-16 — If the axiom does not denote informal text then it is a boolean axiom and "equals true" is inserted before it is transformed

insert-equals-true(*axiom*)(*dict*) $\triangleq$ (3.4.7.1.11)

```
1  if is-Equation0(axiom) then
2    axiom
3   else
4    (let truenm = mk-Name0("TRUE", nil) in
5     let trueid = mk-Id0(⟨hd dict(SCOPEUNIT)⟩, truenm) in
6     mk-Equation0(axiom, trueid))
```

type : $Unquantequation_0 \rightarrow Dict \rightarrow Equation_0$

Objective If an axiom is a boolean term, then construct an AS_0 equation, that is define the term to equals true

Parameters

axiom	The axiom to be tested upon

Result The possibly modified axiom

Algorithm

Line 1	If the axiom is an equation then it is all right else
Line 4	It is a term then construct the name for the boolean sort.
Line 5	Construct the identifier for the boolean sort. The qualifier is the first element in the *Qual* SCOPEUNIT which denotes the current scopeunit. The first element denotes the system level.
Line 6	Construct an equation from the boolean term and the true literal identifier

transform-equation(mk-$Equation_0$(*term1*, *term2*), *context*)(*dict*) $\triangleq$ (3.4.7.1.12)

```
1  (let (as1term1, as1term2, dict') = transform-lhs-rhs(term1, term2, context)(dict) in
2   build-quant-equation(mk-Unquantified-equation1(as1term1, as1term2))(dict'))
```

type : $Equation_0\ Context \rightarrow Dict \rightarrow Equations_1$

Objective Transform a simple unquantified AS_0 equation into an AS_1 equation

Parameters

term1,term2	The two terms in the equation to be transformed
context	See *transform-axioms*

Result The constructed AS_1 equation

Algorithm

Line 1	Transform the two terms. New *ValueidD* descriptors in *dict'* (see the definition of *Dict*) contains information of all the value identifiers used in the two terms.
Line 2	Construct an unquantified equation from the two AS_1 terms and enclose the equation by some quantification (if there are any *ValueidD* descriptors in *dict'*)

transform-lhs-rhs(*term1*, *term2*, *context*)(*dict*) $\triangleq$ (3.4.7.1.13)

```
 1  (let totset = all-visible-sorts(dict) in
 2   let (,tset1,) =
 3       if is-Errorterm0(term1)
 4         then (...,totset,...)
 5         else transform-term(term1, context, totset)(dict),
 6       (,tset2,) =
 7       if is-Errorterm0(term2)
 8         then (...,totset,...)
 9         else transform-term(term2, context, totset)(dict) in
10   let tqual = if card (tset1 ∩ tset2) = 1 then
11                  (let tq ∈ (tset1 ∩ tset2) in
12                   tq)
13                 else
14                  exit("§4.2.3: The two terms in equation must be of the same sort") in
15   let (as1 term1, , d) =
16       if is-Errorterm0(term1)
17         then (mk-Error-term1(), ..., dict)
18         else transform-term(term1, context, {tqual})(dict) in
19   let (as1 term2, , d') =
20       if is-Errorterm0(term2)
21         then (mk-Error-term1(), ..., d)
22         else transform-term(term2, context, {tqual})(d) in
23   (as1 term1, as1 term2, d'))
```

type: $Term_0\ Term_0\ Context \rightarrow Dict \rightarrow Term_1\ Term_1\ Dict$

Objective Transform two AS_0 terms from an equations into two AS_1 terms

Parameters

term1,term2	The two terms to be transformed
context	See *transform-axioms*

Result The two constructed AS_1 terms and a *Dict* including *ValueidD* descriptors of the value identifiers occurring in the two terms

Algorithm

Line 1	Extract the *Qual* set containing all the visible sorts
Line 2-9	Transform the two terms in order to determine the set of legal sorts for the terms. *tset1* denotes the legal sorts for *term1* and *tset2* denotes the legal sorts for *term2*. If one of the terms is the error term then all visible sorts are legal for that term.
Line 10-14	Determine the sort of the equation. There must exist exactly one such sort i.e. the two sets must have one sort in common.
Line 15-23	If one of the terms is the error term then return the AS_1 error term for that term else transform it (line 18 and line 22), given the legal sort as argument (*tqual*). The *Dict* returned from *transform-term* for the first term (line 18) is used when the second term is transformed (line 22) such that the *ValueidD* descriptors are not created twice.

build-quant-equation(*equation*$_1$)(*dict*) $\triangleq$ **(3.4.7.1.14)**

```
1  if (∃q ∈ dom dict)(is-ValueidD(dict(q)) ∧ ¬s-Explicit(dict(q))) then
2    (let q ∈ dom dict be s.t. is-ValueidD(dict(q)) ∧ s-Mapvalue(dict(q)) = nil ∧ ¬s-Explicit(dict(q)) in
3     let mk-ValueidD(, tqs, ) = dict(q) in
4     let tq ∈ tqs in
5     let (,nm) = q[len q] in
6     let equation' = build-quant-equation(equation1)(dict \ {q}) in
7     let as1 tid = make-as1-identifier(tq)(dict) in
8     mk-Quantified-equations1({name-to-name1(nm)}, as1 tid, {equation'}))
9   else
10    equation1
```

type : *Equation*$_1$ $\rightarrow$ *Dict* $\rightarrow$ *Equation*$_1$

Objective Enclose an AS_1 equation by a quantification for every value name found in *dict*

Parameters

equation$_1$	The AS_1 equation to be quantified. As the function is recursive, *equation*$_1$ may be an equation which already is quantified

Algorithm

Line 1	If there exist an *ValueidD* descriptor (value identifier descriptor) in the *Dict* and it is not introduced by explicit quantification then
Line 2	Denote the *Qual* of the value identifier by *q*.
Line 3	Decompose the value identifier descriptor. The set of sorts *tqs* contains at this stage only one sort.
Line 4	Denote the sort of the value identifier by *tq*.
Line 5	Extract the name of the value identifier.
Line 6	Construct a new equation by enclosing the old equation by quantification corresponding to the rest of the value identifiers.
Line 7	Construct the AS_1 identifier from the value identifier *Qual*.
Line 8	Return an equation quantified with an AS_1 name and which contains the new equation

transform-quantequation(mk-*Quantequation*$_0$(*vl*, *tid*, *axl*), *context*)(*dict*) $\triangleq$ **(3.4.7.1.15)**

```
1  (let tqual = get-visible-qual(tid, TYPE)(dict) in
2   let qualset = {dict(SCOPEUNIT) ⌢ ⟨(VALUE, vl[i])⟩ | 1 ≤ i ≤ len vl} in
3   let dict' = dict + [q ↦ mk-ValueidD(nil, {tqual}, true) |
4                       q ∈ qualset] in
5   let as1 ax = transform-axioms(axl, context)(dict') in
6   let as1 tid = make-as1-identifier(tqual)(dict) in
7   let as1 vl = {name-to-name1(vl[i]) | 1 ≤ i ≤ len vl} in
8   mk-Quantified-equations1(as1 vl, as1 tid, as1 ax))
```

type : *Quantequation*$_0$ *Context* $\rightarrow$ *Dict* $\rightarrow$ *Quantified-equations*$_1$

Objective Transform a quantified equation into AS_1

Parameters

vl,tid,axl	The value name list, their sort and the axiom list contained in the quantified equation
context	See *transform-axioms*

Result The AS_1 quantified equation

Algorithm

Line 1	Construct the *Qual* of the sort (*tid*)
Line 2	Construct the *Quals* of the value identifiers corresponding to the value names (*vl*).
Line 3-4	Update *Dict* to include the value identifier descriptors. **nil** indicates that its not a literal value identifiers (mappings), {*tqual*} indicates that the only allowed sort for the value identifiers is *tqual* (in implicit quantification), the set may contain several elements and *true* indicates that the value identifiers are introduced by explicit quantification.
Line 5	Transform the axioms in the quantified equation into AS_1.
Line 6	Construct an AS_1 identifier for the sort *Qual*.
Line 7	Construct an AS_1 name set containing the value names.
Line 8	Return a constructed AS_1 quantified equation

transform-condequation(mk-*Condequation*$_0$(*eql*, *eq*), *context*)(*dict*) $\triangleq$ **(3.4.7.1.16)**

```
1  (let (eql1, dict') = transform-restrictions(eql, context)(dict) in
2   let mk-Equation0(term1, term2) = insert-equals-true(eq)(dict) in
3   let (as1term1, as1term2, dict'') = transform-lhs-rhs(term1, term2, context)(dict') in
4   let eq1 = mk-Unquantified-equation1(as1term1, as1term2) in
5   build-quant-equation(mk-Conditional-equation1(eql1, eq1))(dict''))
```

type: *Condequation*$_0$ *Context* $\rightarrow$ *Dict* $\rightarrow$ *Equation*$_1$

Objective Transform a conditional equation into AS_1

Parameters

eql,eq	The restriction and the restricted equations in the conditional equation

Result The constructed AS_1 conditional equation

Algorithm

Line 1	Transform the restrictions.
Line 2	Decompose the AS_0 equation.
Line 3	Transform the two terms in the AS_0 restricted equation. Note that *dict'* is used in the transformation of the restricted equations such that the implicit value identifiers in the two parts are the same.
Line 4	Construct an AS_1 equation.
Line 5	Enclose the constructed equation by quantification (if *dict''* contains any *ValueidD* descriptors).

transform-restrictions(*eql*, *context*)(*dict*) $\triangleq$ **(3.4.7.1.17)**

```
1  (if eql = ⟨⟩ then
2     ({}, dict)
3    else
4     (let mk-Equation0(term1, term2) = insert-equals-true(hd eql)(dict) in
5      if is-Errorterm0(term1) ∨ is-Errorterm0(term2) then
6        exit("§5.4.1.7: Error term is part of restriction")
7       else
8        (let (as1term1, as1term2, dict') = transform-lhs-rhs(term1, term2, context)(dict) in
9         let (eqlrest, dictrest) = transform-restrictions(tl eql, context)(dict') in
10        (eqlrest ∪ {mk-Unquantified-equation1(as1term1, as1term2)}, dictrest))))
```

type: *Unquantequation*$_0$* *Context* $\rightarrow$ *Dict* $\rightarrow$ *Equation*$_1$-**set** *Dict*

Objective Transform a list of restrictions into a set of AS_1 restrictions by traversing recursively through the list and by collecting information in *Dict* of the value identifiers. The function is similar to *transform-equation*, but applied on a list of equations instead.

$transform\text{-}mapping(\mathbf{mk}\text{-}Mappingaxiom_0(vl, tid, axl))(dict) \triangleq$ (3.4.7.1.18)

```
1  (let tqual = get-visible-qual(tid, TYPE)(dict) in
2   let qualset = {dict(SCOPEUNIT) ⌢ ⟨(VALUE, vl[i])⟩ | 1 ≤ i ≤ len vl} in
3   let litset = {qual ∈ dom dict | is-LiteralD(dict(qual)) ∧ s-Sortqual(dict(qual)) = tqual} in
4   transform-mappingaxioms(qualset, tqual, litset, axl)(dict))
```

type : $Mappingaxiom_0 \rightarrow Dict \rightarrow Equations_1$

Objective Transform a mapping axiom into a set of AS_1 equations

Parameters

vl	The value name list in the mapping axiom
tid	The sort of the value names
axl	The axioms contained in the mapping axiom

Result The transformed AS_1 equation

Algorithm

Line 1	Extract the *Qual* of the sort.
Line 2	Construct the set of *Quals* for the value names.
Line 3	Extract the set of literal *Quals* defined for the sort *tqual.*
Line 4	Do the transformation

$transform\text{-}mappingaxioms(qualset, tqual, litset, axl)(dict) \triangleq$ (3.4.7.1.19)

```
1  if qualset = {} then
2    {}
3   else
4    (let qual ∈ qualset in
5     union {transform-axioms(axl, MAPPING)(dict + [qual ↦ mk-ValueidD(litqual, {tqual}, true)]) |
6            litqual ∈ litset} ∪
7     transform-mappingaxioms(qualset \ {qual}, tqual, litset, axl)(dict))
```

type : *Qual*-**set** *Sortqual Qual*-**set** $Axiom_0^* \rightarrow Dict \rightarrow Equations_1$

Objective With a given set of value identifiers, their sort and the literals defined for that sort, expand a set of equations by replacing all the value identifiers in the axioms by a literal

Parameters

qualset	A set of *Quals* representing the value identifiers
tqual	A *Qual* representing the identifier of their sort
litset	A set of *Quals* representing the literals defined for *tqual*
axl	The axioms to be expanded

Result A set of AS_1 Equations where the value identifiers corresponding to *qualset* have been replaced

Algorithm

Line 1	When through, return the empty set of equations.
Line 4	Take the next value identifier in the set.

Line 5-6 Construct the set of equations which is the union of all those set of equations where the value identifier is replaced by a particular literal in *litset*.

Line 7 Return this constructed set of equations, unified with the equations constructed by replacing the rest of the value identifiers

add-equality(*tqual*)(*dict*) $\triangleq$ (3.4.7.1.20)

```
 1  (let bqual = get-predef-sort("BOOLEAN")(dict) in
 2   let bid = as0-id(bqual) in
 3   let tid = as0-id(tqual) in
 4   let opl = ⟨mk-Opspec0(EQ, ⟨tid, tid⟩, bid), mk-Opspec0(NE, ⟨tid, tid⟩, bid)⟩ in
 5   let anm = create-unique-name(),
 6       bnm = create-unique-name(),
 7       cnm = create-unique-name(),
 8       trueid = mk-Id0(bqual, mk-Name0("TRUE", nil)) in
 9   let aid = mk-Id0(⟨⟩, anm),
10       bid = mk-Id0(⟨⟩, bnm),
11       cid = mk-Id0(⟨⟩, cnm) in
12   let eq1 = mk-Equation0(mk-Infixterm0(aid, EQ, aid), trueid),
13       eq2 = mk-Equation0(mk-Infixterm0(aid, EQ, bid), mk-Infixterm0(bid, EQ, aid)),
14       eq3 = mk-Equation0(mk-Infixterm0(mk-Infixterm0(mk-Infixterm0(aid, EQ, bid), AND,
15                                                    mk-Infixterm0(bid, EQ, cid)),
16                                              IMPLY, mk-Infixterm0(aid, EQ, cid)), trueid),
17       eq4 = mk-Equation0(mk-Infixterm0(aid, NE, bid),
18                          mk-Monadterm0(NOT, mk-Infixterm0(aid, EQ, bid))),
19       eq5 = mk-Condequation0(⟨mk-Equation0(mk-Infixterm0(aid, EQ, bid), trueid)⟩,
20                              mk-Equation0(aid, bid)) in
21   (opl, ⟨mk-Quantequation0(⟨anm, bnm, cnm⟩, tid, ⟨eq1, eq2, eq3, eq4, eq5⟩)⟩))))
```

type: *Sortqual* → *Dict* → *Opspec*$_0^+$ *Axiom*$_0^+$

Objective For a given sort, construct the AS$_0$ equality properties as defined in §5.4.1.4 of Z.100

Parameters

tqual The *Qual* of the sort

Result An AS$_0$ list of operator definitions and an AS$_0$ list of axioms

Algorithm

Line 1-3 Construct the AS$_0$ (qualified) identifiers of the boolean sort and of *tqual*.

Line 4 Construct the definitions:

```
"="  : tqual, tqual -> boolean
"/=" : tqual, tqual -> boolean
```

Line 5-7 Create some distinct names to be used in the quantification.

Line 8 Construct the qualified AS$_0$ identifier of the boolean literal TRUE.

Line 9-11 Construct AS$_0$ identifiers for the value names.

Line 12-20 Construct the five axioms (*eq1* - *eq5*):

```
aid = aid                      == true;
aid = bid                      == bid = aid;
(aid = bid) and (bid = cid) =>
                    aid = cid  == true;
aid /= bid                     == not(aid = bid);
aid = bid == true ==>  aid     == bid;
```

Line 21 Return the constructed operator definitions and axioms

add-ordering(mk-*Properties*$_0$(*lit*, *oplist*, *axioms*, *map*, *init*), *tqual*)(*dict*) $\triangleq$ (3.4.7.1.21)

```
 1  (if (∃op ∈ elems oplist)(is-Ordering0(op)) then
 2     (let oplist' = ⟨oplist[i] | 1 ≤ i ≤ len oplist ∧ is-Opspec0(oplist[i])⟩ in
 3      if len oplist = len oplist' − 1 then
 4        (let ordaxioms = as0-ordering-axioms(tqual)(dict),
 5             order = if len lit ≤ 1 then ⟨⟩ else as0-order(hd lit, tl lit, tqual),
 6             ordtyping = as0-ordering-typing(tqual)(dict) in
 7         mk-Properties0(lit, ordtyping ⌢ oplist', ordaxioms ⌢ axioms ⌢ order, map, init))
 8       else
 9        exit("§2.2.2: Ordering operators are multiple defined"))
10    else
11     mk-Properties0(lit, oplist, axioms, map, init))
```

type : *Properties*$_0$ *Sortqual* $\rightarrow$ *Dict* $\rightarrow$ *Properties*$_0$

Objective Expand the ordering directive in some properties into operator definitions and axioms as defined in §5.4.1.8 of Z.100

Parameters The properties consist of

lit	Literal definitions
oplist	Operator definitions
axioms	Axioms
map	Mapping axioms
init	The default value
tqual	Denotes the sort defining the properties

Result The modified properties

Algorithm

Line 1	If there exist an ordering directive in the operator definitions then
Line 2	Construct an operator definition list where the ordering directives have been removed.
Line 3	There must exist only one such directive in the operator list.
Line 4	Construct the AS_0 properties of the ordering operators.
Line 5	If there are any literals defined for the sort then construct the axioms reflecting the ordering of the literals.
Line 6	Construct the definitions for the ordering operators.
Line 7	Return the modified properties

as$_0$*-ordering-axioms*(*squal*)(*dict*) $\triangleq$ **(3.4.7.1.22)**

```
 1  (let bqual = get-predef-sort("BOOLEAN")(dict) in
 2   let falseid = mk-Id0(bqual, mk-Name0("FALSE", nil)) in
 3   let sid = as0-id(squal) in
 4   let anm = create-unique-name(),
 5       bnm = create-unique-name(),
 6       cnm = create-unique-name() in
 7   let aid = mk-Id0(⟨⟩, anm),
 8       bid = mk-Id0(⟨⟩, bnm),
 9       cid = mk-Id0(⟨⟩, cnm) in
10   let axioml =
11       ⟨mk-Equation0(mk-Infixterm0(aid, LT, aid), falseid),
12        mk-Equation0(mk-Infixterm0(aid, LT, bid), mk-Infixterm0(bid, GT, aid)),
13        mk-Equation0(mk-Infixterm0(aid, LE, bid), mk-Infixterm0(mk-Infixterm0(aid, LT, bid),
14                                                                OR,
15                                                                mk-Infixterm0(aid, EQ, bid))),
16        mk-Equation0(mk-Infixterm0(aid, GE, bid), mk-Infixterm0(mk-Infixterm0(aid, LE, bid),
17                                                                OR,
18                                                                mk-Infixterm0(aid, EQ, bid))),
19        mk-Infixterm0(mk-Infixterm0(aid, LT, bid),
20                      IMPLY,
21                      mk-Monadexpr0(NOT, mk-Infixterm0(aid, LT, bid))),
22        mk-Infixterm0(mk-Infixterm0(mk-Infixterm0(aid, LT, bid),
23                                    AND, mk-Infixterm0(bid, LT, cid)),
24                      IMPLY,
25                      mk-Infixterm0(aid, LT, cid))⟩ in
26   ⟨mk-Quantequation0(⟨anm, bnm, cnm⟩, sid, axioml)⟩)
```

type: *Sortqual* → *Dict* → *Axiom*$_0$

Objective

Construct the AS$_0$ axiom reflecting the properties of the ordering operators i.e. :

```
FOR ALL aid,bid,cid IN sid
(aid < aid  == False;
 aid < bid  == bid > aid;
 aid <= bid == (aid < bid) OR (aid = bid);
 aid >= bid == (aid > bid) OR (aid = bid);
 aid < bid  => not(bid < aid);
 aid < bid  and bid < cid => aid < cid);
```

Algorithm

Line 11 Construct the 1'th equation.

Line 12 Construct the 2'th equation.

Line 13 Construct the 3'th equation.

Line 16 Construct the 4'th equation (i.e. a boolean axiom).

Line 19 Construct the 5'th equation (i.e. a boolean axiom).

Line 22 Construct and return the quantified equation which contains the five equations.

as_0-*ordering-typing*(*tqual*)(*dict*) $\triangleq$ (3.4.7.1.23)

```
1  (let tid = as0-id(tqual),
2       bid = as0-id(get-predef-sort("BOOLEAN")(dict)) in
3   ⟨mk-Opspec0(LT, ⟨tid, tid⟩, bid),
4    mk-Opspec0(GT, ⟨tid, tid⟩, bid),
5    mk-Opspec0(LE, ⟨tid, tid⟩, bid),
6    mk-Opspec0(GE, ⟨tid, tid⟩, bid)⟩)
```

type : $Name_0 \rightarrow Dict \rightarrow Opspec_0^+$

Objective Construct the AS_0 definitions of the ordering operators i.e. :

```
"<"  : tid, tid -> bid;
">"  : tid, tid -> bid;
"<=" : tid, tid -> bid;
">=" : tid, tid -> bid;
```

as_0-*order*(*lit*, *litlist*, *squal*) $\triangleq$ (3.4.7.1.24)

```
1  (if litlist = ⟨⟩ then
2     ⟨⟩
3   else
4    (if is-Nmclass0(lit) then
5       as0-order(hd litlist, tl litlist, squal)
6     else
7      if is-Nmclass0(hd litlist) then
8        as0-order(lit, tl litlist, squal)
9      else
10      (let id1 = if is-Name0(lit) then
11                   mk-Id0(squal, lit)
12                 else
13                   mk-Stringterm0(squal, lit),
14           id2 = if is-Name0(hd litlist) then
15                   mk-Id0(squal, hd litlist)
16                 else
17                   mk-Stringterm0(squal, hd litlist) in
18       ⟨mk-Infixterm0(id1, LT, id2)⟩ ⌢
19       as0-order(hd litlist, tl litlist, squal))))
```

type : $Literal_0\ Literal_0^*\ Sortqual \rightarrow Axiom_0^*$

Objective Construct the axioms reflecting the ordering of the literals for a sort

Parameters

lit	The first literal in the literal list
litlist	The rest of the literal list (the function is recursive)
squal	The *Qual* of the sort defining the literals

Result A list of axioms on the form :

```
id1 < id2
```

Algorithm

Line 1	When through, return nothing.
Line 4-5	If the literal in hand is a name class then skip it over.
Line 7-8	If the next literal in the literal list is a name class then skip it over.
Line 10-14	Construct the AS_0 identifier of the literal in hand and of the next literal in the list. Note that the full qualifier is present in order to avoid ambiguity.

Line 18-19 Return the boolean axiom for this literal concatenated with the boolean axioms for the rest of the literals

transform-typing(*oplist*)(*dict*) $\triangleq$ (3.4.7.1.25)

```
1  (if oplist = ⟨⟩ then
2     ({}, [])
3   else
4    (let mk-Opspec0(nm, opsortl, ressort) = hd oplist in
5     let (oprest, drest) = transform-typing(tl oplist)(dict) in
6     let tqual = get-visible-qual(ressort, TYPE)(dict) in
7     let pqual = get-parent(tqual)(dict) in
8     let as1id = make-as1-identifier(tqual)(dict) in
9     let arglist = ⟨get-visible-qual(opsortl[i], TYPE)(dict) | 1 ≤ i ≤ len opsortl⟩ in
10    let parglist = ⟨get-parent(arglist[i])(dict) | 1 ≤ i ≤ len arglist⟩ in
11    let opparml = ⟨make-as1-identifier(arglist[i])(dict) | 1 ≤ i ≤ len arglist⟩ in
12    let opqual = dict(SCOPEUNIT) ⌢ ⟨(OPERATOR, (nm, parglist, pqual))⟩ in
13    let newnm = create-unique-name() in
14    let newqual = dict(SCOPEUNIT) ⌢ ⟨(OPERATOR, (newnm, parglist, pqual))⟩ in
15    let descr = mk-OperatorD(arglist, tqual, newqual, true) in
16    let as1tree = mk-Operator-signature1(name-to-name1(newnm), opparml, as1id) in
17    if opqual ∈ dom drest then
18      exit("§2.2.2: Multiple appearence of operator definition")
19     else
20      ({as1tree} ∪ oprest, drest + [opqual ↦ descr])))
```

type: $Opspec_0^* \rightarrow Dict \rightarrow Operator\text{-}signature_1\text{-}\mathbf{set}\ Dict$

Objective Transform a list of operator definitions into AS_1 and update *Dict* to include descriptors for the operators

Parameters

oplist The list of AS_0 operator definitions

Result The AS_1 operator definitions and the updated *dict*

Algorithm

Line 1 When through, return empty objects (the function is recursive).

Line 4 Decompose the first operator definition in the list.

Line 5 Transform the rest of the operator definitions.

Line 6-7 Extract the *Qual* of the result sort and fetch the parent sort (*pqual*) in the case of a syntype.

Line 8 Construct the AS_1 identifier of the result sort.

Line 9-10 Extract the *Quals* of the argument sorts and fetch the parent sorts (*parglist*) in the case of any syntypes.

Line 11 Construct the AS_1 identifier list of the argument sorts.

Line 12 Construct the *Quals* of the operator. Note that the *Operatorqual-elem* (see the domain definition) contains the parent sorts (no syntypes) such that uniqueness of operators can be verified by inspecting the operator *Quals*.

Line 13 Construct a unique name for the operator. (Unique names are generated for all operators such that no overloading occurs in AS_1).

Line 14 Construct the operator *Qual* which is used for transforming the applied occurrences of the operator into AS_1.

Line 15 Construct the *Dict* descriptor for the operator. This time, the arguments and the result can be syntypes.

Line 16 Construct the AS_1 signature of the operator.

Line 17	If the *Qual* of the operator occurs in the *Dict* contribution for the rest of the operators then two operators with the same name, the same argument sorts and the same result are defined within the same sort definition.
Line 20	Return the signature for this operator unified with the signatures for the rest of the operators and return the *Dict* contributions for this operator and for the rest of the operators

3.4.7.2 Struct Definitions

$$\textit{transform-struc}(nm, \textbf{mk-}\textit{Struc}_0(fl), \textbf{mk-}\textit{Properties}_0(l, o, a, am, init))(dict) \triangleq \quad (3.4.7.2.1)$$

```
1  (let sid = mk-Id0(dict(SCOPEUNIT), nm) in
2   let (o', a', ) = transform-fields(sid, fl, {}, ⟨⟩) in
3   let prop' = mk-Properties0(l, o ⌢ o', a ⌢ a', am, init) in
4   transform-sortdef(nm, prop', nil)(dict))
```

type : $Name_0\ Struc_0\ Properties_0 \rightarrow Dict \rightarrow Dict$

Objective Transform a STRUCT sort definition into AS_1

Parameters

nm	The name of the sort being defined
fl	The field list
l,o,a,am,init	The literals, the operators, the axioms, the mapping axioms and the default value specified in the ADDING construct

Result The *Dict* updated to include descriptors for the literals and operators and updated to include the generated AS_1 objects in its DATATYPEDEF entry

Algorithm

Line 1	Construct the AS_0 identifier of the sort being defined.
Line 2	Generate the AS_0 operator definitions and axioms implied by the struct definition.
Line 3	Construct the AS_0 properties containing the implied properties and the additional properties.
Line 4	Transform those properties in the usual way

transform-fields(*tid*, *flist*, *fset*, *sidlist*) $\triangleq$ (3.4.7.2.2)

```
1  (if flist = ⟨⟩ then
2     (⟨mk-Opspec0(mk-Name0("MAKE", EXCLAMATION), sidlist, tid)⟩, ⟨⟩, len sidlist)
3   else
4    (let mk-Fieldspec0(fnml, fid) = hd flist in
5     let nm = hd fnml in
6     if nm ∈ fset then
7       exit("§5.4.1.10: Two structure fields have the same name")
8      else
9       (let flist' = if tl fnml = ⟨⟩ then
10                       tl flist
11                    else
12                       ⟨mk-Fieldspec0(tl fnml, fid)⟩ ⌢ tl flist in
13        let (as0oprest, as0axrest, argnum) =
14            transform-fields(tid, flist', fset ∪ {nm}, sidlist ⌢ fid) in
15        let (as0op, as0ax) = transform-field(tid, fid, nm, argnum, len sidlist + 1) in
16        (as0op ⌢ as0oprest, as0ax ⌢ as0axrest, argnum))))
```

type: $Sortid_0$ $Fieldspec_0^*$ $Fieldname_0$-set $Sortid_0^*$ $\rightarrow$ Op_0^* $Axiom_0^*$ N_1

Objective Transform a field list from an AS_0 struct definition into a list of operator definitions and a list of axioms

Parameters

tid	The AS_0 identifier of the sort defining the struct
flist	The field list
fset	The set of already used field names (the function is recursive)
sidlist	The current list of sorts for the fields. This list is used for constructing the signature of the MAKE! operator

Result The AS_0 operator definitions, the axioms implied by the field list and the complete number of fields. This number is only used internal in the function

Algorithm

Line 1-2	When through then *sidlist* is complete and the signature of the MAKE! operator can be constructed. The length of *sidlist* is also returned such that the axioms including the MAKE! operator, constructed at the other recursion levels, can be made properly.
Line 4	Decompose the first field specification.
Line 5	Let *nm* denote the first name in the field name list of the first field specification.
Line 6	If the name already has been used then it is an error.
Line 9	Construct a new field list where the name in hand is excluded.
Line 13	Transform the rest of the fields.
Line 15	Construct the operator definitions and the axioms for the field in hand
Line 16	Return the properties for the field in hand concatenated with the properties for the rest of the fields

transform-field(*tid*, *fid*, *fnm*, *totnum*, *index*) $\triangleq$ (3.4.7.2.3)

```
1  (let (modop, modax) = as0-modify-properties(tid, fid, fnm, totnum, index),
2       (extop, extax) = as0-extract-properties(tid, fid, fnm, totnum, index) in
3   (⟨modop ⌢ extop⟩, ⟨modax ⌢ extax⟩))
```

type: $Sortid_0$ $Sortid_0$ $Fieldname_0$ N_1 N_1 $\rightarrow$ Op_0^* $Axiom_0^*$

Objective Construct the implied operator definitions and axioms attached to a field name

Parameters

tid	The AS_0 identifier of the sort defining the struct
fid	The AS_0 identifier of the field sort
fnm	The field name
totnum	The complete number of fields for the sort *tid*
index	The position of the field in the argument list of the MAKE! operator attached to *tid*

Result The Operator definitions and axioms attached to the field *fnm*

Algorithm

Line 1	Construct the operator definition and axiom for the modify operator attached to the field.
Line 2	Construct the operator definition and axiom for the extract operator attached to the field.
Line 3	Return the two operator definitions and the two axioms

as_0-*modify-properties*(*tid*, *fid*, **mk**-$Name_0$(*fst*,), *totlistlength*, *index*) $\triangleq$ **(3.4.7.2.4)**

```
1  (let mnm = mk-Name0(fst ⌢ "MODIFY", EXCLAMATION) in
2   let mk-Id0(qualifier, tnm) = tid in
3   let tqual = qualifier ⌢ ⟨(TYPE, tnm)⟩ in
4   let opspec = mk-Opspec0(mnm, ⟨tid, fid⟩, tid) in
5   let fvarid = mk-Id0(⟨⟩, create-unique-name()) in
6   let makearglist = ⟨mk-Id0(⟨⟩, create-unique-name()) | 1 ≤ i ≤ totlistlength⟩ in
7   let makearglist' = ⟨makearglist[i] | 1 ≤ i < index⟩ ⌢
8       ⟨fvarid⟩ ⌢
9       ⟨makearglist[i] | index < i ≤ totlistlength⟩ in
10  let makenm = mk-Name0("MAKE", EXCLAMATION) in
11  let makeop = mk-Operatorterm0(mk-Id0(tqual, makenm), makearglist),
12      makeop' = mk-Operatorterm0(mk-Id0(tqual, makenm), makearglist') in
13  let lterm = mk-Operatorterm0(mk-Id0(tqual, mnm), ⟨makeop, fvarid⟩) in
14  let ax = mk-Equation0(lterm, makeop') in
15  (opspec, ax))
```

type : $Sortid_0\ Sortid_0\ Fieldname_0\ N_1\ N_1 \rightarrow Op_0\ Axiom_0$

Objective Construct the AS_0 properties for the modify operator attached to a struct field

Parameters

tid	The identifier of the sort defining the struct
fid	The identifier of the field sort
fst	The spelling of the field name
totlistlength	The number of fields defined for the sort *tid*
index	The position of the field in the argument list of the MAKE! operator attached to *tid*

Result The implied operator definition and axiom as defined in §5.4.1.10 of Z.100

Algorithm

Line 1	Construct the name of the modify operator by concatenating the spelling of the field name by the string "MODIFY".

Line 2-3	Construct the qualifier (*tqual*) denoting the sort *tid*. The MAKE! operator is qualified by *tqual* in order to avoid ambiguity.
Line 4	Construct the operator definition for the field modify operator.
Line 5-6	Construct a distinct value identifier (*fvarid*) and create a list of distinct value identifiers to be used as argument to the (left-hand) MAKE! operator (see the example in Z.100 §5.4.1.10).
Line 6-9	Modify the argument list by replacing the value identifier at position *index* by *fvarid*. This argument list is used on the right hand side of the axiom.
Line 10	Construct the $Name_0$ of the MAKE! operator.
Line 11	Make the AS_0 operator application for the left hand MAKE! operator.
Line 12	Make the AS_0 operator application for the right hand side term.
Line 13	Construct the left hand side term.
Line 14-15	Return the operator definition and the constructed axiom

as_0-*extract-properties*(*tid*, *fid*, mk-$Name_0$(*fst*,), *totlistlength*, *index*) $\triangleq$ **(3.4.7.2.5)**

```
 1  (let enm = mk-Name0(fst ⌢ "EXTRACT", EXCLAMATION) in
 2   let mk-Id0(qualifier, tnm) = tid in
 3   let tqual = qualifier ⌢ ⟨(TYPE, tnm)⟩ in
 4   let opspec = mk-Opspec0(enm, ⟨tid⟩, fid) in
 5   let makearglist = ⟨mk-Id0(⟨⟩, create-unique-name()) | 1 ≤ i ≤ totlistlength⟩ in
 6   let makenm = mk-Name0("MAKE", EXCLAMATION) in
 7   let makeop = mk-Operatorterm0(mk-Id0(tqual, makenm), makearglist) in
 8   let lterm = mk-Operatorterm0(mk-Id0(tqual, enm), ⟨makeop⟩) in
 9   let ax = mk-Equation0(lterm, makearglist[index]) in
10   (opspec, ax))
```

type: $Sortid_0\ Sortid_0\ Fieldname_0\ N_1\ N_1 \rightarrow Op_0\ Axiom_0$

Objective Construct the AS_0 properties for the extract operator attached to a struct field

Parameters

tid	The identifier of the sort defining the struct
fid	The identifier of the field sort
fst	The spelling of the field name
totlistlength	The number of fields defined for the sort *tid*
index	The position of the field in the argument list of the MAKE! operator attached to *tid*

Result The implied operator definition and axiom as defined in §5.4.1.10 of Z.100

Algorithm

Line 1	Construct the name of the extract operator by concatenating the spelling of the field name by the string "EXTRACT".
Line 2-3	Construct the qualifier (*tqual*) denoting the sort *tid*.
Line 4	Construct the operator definition for the field extract operator.
Line 5-6	Construct a distinct value identifier (*fvarid*) and create a list of distinct value identifiers to be used as argument to the (left-hand) MAKE! operator (see the example in Z.100 §5.4.1.10).
Line 6-7	Construct the left hand side MAKE! operator application.
Line 8	Construct the left hand side term.
Line 10-11	Return the constructed operator definition and axiom

3.4.8 Timer Definitions

transform-timerdef(mk-*Timerdef*$_0$(*vlist*))(*dict*) $\triangleq$ (3.4.8.1)

```
1  if (∃squal ∈ dom dict)(is-ServiceD(dict(squal)) ∧ get-sur(squal) = dict(SCOPEUNIT)) then
2    exit("§4.10.2: Process contains both service definitions and timer definitions")
3   else
4    (let mk-Timerelem0(tnm, sortlist) = hd vlist in
5     let tqual = dict(SCOPEUNIT) ⌢ ⟨(SIGNAL, tnm)⟩ in
6     let unm = if is-ServiceD(dict(dict(SCOPEUNIT))) then create-unique-name() else tnm in
7     let nqual = dict(SCOPEUNIT) ⌢ ⟨(SIGNAL, unm)⟩ in
8     let squallist = ⟨get-visible-qual(sortlist[i], TYPE)(dict) | i ∈ ind sortlist⟩ in
9     let as1idlist = ⟨make-as1-identifier(squallist[i])(dict) | i ∈ ind squallist⟩ in
10    let as1tree = mk-Timer-definition1(name-to-name1(unm), as1idlist),
11        delem = [tqual ↦ mk-TimerD(squallist, nqual) | 1 ≤ i ≤ len squallist] in
12    if tl vlist = ⟨⟩ then
13      (⟨as1tree⟩, delem)
14     else
15      (let (as1rest, drest) = transform-timerdef(mk-Timerdef0(tl vlist))(dict) in
16       ({as1tree} ∪ as1rest, delem + drest)))
```

type: $Timerdef_0^+ \rightarrow Dict \rightarrow Timer\text{-}definition_1$-set $Dict$

Objective Transform a timer definition into AS_1

Parameters

vlist	The list of (elementary) timer definitions contained in the (composite) timer definition

Result A set of AS_1 timer definitions and the *Dict* contribution containing the descriptors for the timers

Algorithm

Line 1-2	The surrounding process must not contain any services.
Line 4	Decompose the next elementary timer definition (the function is recursive).
Line 5	Construct the *Qual* for the timer.
Line 6-7	Construct the unique name and the unique *Qual* which is used for deriving the AS_1 identifier (see the definition of *Newqual*)
Line 8-9	Construct the list of sort *Quals* attached to the timer and construct the corresponding AS_1 sort identifier list.
Line 10	Construct the AS_1 timer definition for the timer in hand
Line 11	Construct the *Dict* contribution for the timer in hand
Line 12	If this elementary timer definition is the last element in the list then return the AS_1 definition and the *Dict* contribution else
Line 15-16	Join them with the definitions and *Dict* contribution corresponding to the rest of the elementary timer definitions

3.4.9 Variable Definitions

transform-vardef(mk-*Vardef*$_0$(*revatt*, *expatt*, *vlist*))(*dict*) $\triangleq$ (3.4.9.1)

```
1  (let mk-Vardefelem0(vl, tp, expr) = hd vlist in
2   let tqual = get-visible-qual(tp, TYPE)(dict) in
3   let pq = get-parent(tqual)(dict) in
4   let expr' = if expr = nil then
5                 cases dict(tqual):
6                  (mk-SortD(, , expr1, )     → expr1,
```

```
 7                    mk-SyntypeD(, , expr1, ) → expr1)
 8                 else
 9                  (let (expr1, , ) = transform-expr(expr, CONSTANT, {pq})(dict) in
10                   expr1) in
11  let vnum = len vl in
12  let scope = dict(SCOPEUNIT) in
13  let nmlist = ⟨if is-ServiceD(dict(scope)) then create-unique-name() else vl[i] | 1 ≤ i ≤ vnum⟩ in
14  let quallist = ⟨scope ⌢ ⟨(VALUE, vl[i])⟩ | 1 ≤ i ≤ vnum⟩,
15      newquallist = ⟨scope ⌢ ⟨(VALUE, nmlist[i])⟩ | 1 ≤ i ≤ vnum⟩ in
16  let (decll, dexport) = if expatt = nil then
17                           ({}, [])
18                         else
19                          build-exported-vardef(vl, tqual, expr)(dict) in
20  let (as1 set, delem) =
21      (let as1 tp = make-as1-identifier(tqual)(dict) in
22       let as1 dl = {mk-Variable-definition1(name-to-name1(nmlist[i]), as1 tp, revatt) | i ∈ ind vnum},
23           d = [quallist[i] ↦ mk-VarD(tqual, revatt, expatt, expr', newquallist[i]) | 1 ≤ i ≤ vnum] in
24       (as1 dl, d)) in
25  if ¬is-ProcessD(dict(scope)) ∧ (revatt ≠ nil ∨ expatt ≠ nil) then
26    exit("§2.4.5: Variable in procedure (or service) cannot be EXPORTED or REVEALED")
27   else
28    if card elems vl ≠ len vl then
29      exit("§2.2.2: Two definitions in the same scopeunit use the same name")
30     else
31      if tl vlist = ⟨⟩ then
32        (as1 set ∪ decll, delem + dexport)
33       else
34        (let (as1 rest, drest) =
35             transform-vardef(mk-Vardef0(revatt, expatt, tl vlist))(dict) in
36         if dom delem ∩ dom drest ≠ {} then
37           exit("§2.2.2: Two definitions in the same scopeunit use the same name")
38          else
39           (as1 set ∪ decll ∪ as1 rest, delem + drest + dexport)))
```

type: $Vardef_0 \rightarrow Dict \rightarrow Variable\text{-}definition_1\text{-}\mathbf{set}\ Dict$

Objective Transform a variable definition into AS_1

Parameters The variable definition containing:

revatt	The optional REVEAL attribute
expatt	The optional EXPORT attribute
vlist	The list of (elementary) variable definitions

Result The set of AS_1 variable definitions and a *Dict* contribution containing the descriptors of the defined variables

Algorithm

Line 1	Decompose the first elementary variable definition (the function is recursive).
Line 2	Extract the *Qual* of their sort.
Line 3	Extract the *Qual* of its parent (in the case of a syntype).
Line 4-10	If the initial expression is not specified in the variable definitions then extract the AS_1 default expression for its sort (which also may be **nil**) otherwise transform the expression specified.
Line 11	Let *vnum* denote the number of variable names introduced in the elementary variable definition.
Line 12	Let *scope* denote the *Qual* of the scopeunit defining the variables.
Line 13	Construct new unique names for the variables if they are defined in a service.

Line 14	Construct the variable *Quals* to be used as the *Dict* entries for the variables.
Line 15	Construct the *Quals* to be put into the *Newqual* component of the variable descriptors (they differs from the *Quals* constructed at line 14 if in a service).
Line 16-19	If the variables have the EXPORT attribute then construct the AS_1 definitions (*decll*) for the implicit variables.
Line 20	Construct the AS_1 variable definitions and *Dict* contributions for the variables by:.
Line 20	Constructing the AS_1 identifier of their sort.
Line 21	For each of the variables constructing an AS_1 variable definition containing the name (*nmlist*[*i*]), its sort ($as_1 tp$) and possibly a reveal attribute (*revatt*).
Line 23	For each of the variables, construct a *Dict* descriptor containing the *Qual* of its sort (*tqual*), possibly a reveal attribute (*revatt*), possibly an export attribute (*expatt*), possibly an initial AS_1 expression (*expr'*) and the (possible new) *Qual* to be used in the applied occurrences of the variable.
Line 25-26	Unless the variables are defined at the process level they cannot be EXPORTED or REVEALED.
Line 28	The names in *vl* must be unique.
Line 31	If there are no more elementary variable definitions in the composite variable definition then return the constructed objects else
Line 34-39	Join them with the AS_1 definitions and *Dict* contributions for the rest of the elementary variable definitions provided the names in the variable definition are unique

build-exported-vardef(*varlist*, *tqual*, *expr*)(*dict*) $\triangleq$ (3.4.9.2)

```
1  if varlist = ⟨⟩ then
2    ({}, [])
3   else
4    (let nm = hd varlist in
5     let (vdrest, drest) = build-exported-vardef(tl varlist, tqual, expr)(dict) in
6     let id0 = as0-id(tqual) in
7     let mk-SignalnamesD(impnm, , ) = exportmap((tqual, nm)) in
8     let vardef0 = mk-Vardef0(nil, nil, ⟨mk-Vardefelem0(⟨impnm⟩, id0, expr)⟩) in
9     let (vardef1, dict') = transform-vardef(vardef0)(dict) in
10    (vdrest ∪ vardef1, drest + dict'))
```

type: $Name_0^*$ *Qual* [$Expr_0$] $\rightarrow$ *Dict* $\rightarrow$ *Variable-definition*$_1$-**set** *Dict*

Objective Construct the AS_1 variable definitions for the objects implied by exported variables

Parameters

varlist	The list of variables which are exported
tqual	The *Qual* of the variable sort
expr	The expression denoting the initial value of the variables

Result The AS_1 definitions

Algorithm

Line 1	When through, return nothing (the function is recursive).
Line 4	Let *nm* denote the first (next) name in the variable list.
Line 5	Construct the AS_1 definitions for the rest of the variable list
Line 6	(Re)construct the AS_0 identifier of the variable sort.

Line 7	Decompose the export descriptor (*SignalnamesD*) in order to access the name of the implicit variable.
Line 8	Construct an AS_0 variable definition of that implicit name.
Line 9	Transform the implicit variable definition into AS_1.
Line 10	Return that AS_1 definition joined with the definitions for the rest of the exported variables. The *VarD* descriptor of the implicit variable (contained in *dict'*) is included in the resulting *Dict*.

3.4.10 View Definitions

transform-viewdef(mk-*Viewdef*$_0$(*vars*))(*dict*) $\triangleq$ (3.4.10.1)

```
1  (let mk-Viewdefelem0(varlist, tp) = hd vars in
2   if card elems varlist ≠ len varlist then
3     exit("§2.2.2: Two definitions in the same scopeunit use the same name")
4    else
5     (let tqual = get-visible-qual(tp, TYPE)(dict) in
6      let tqual' = get-parent(tqual)(dict) in
7      let (as1 set, d) = transform-view(varlist, tqual')(dict) in
8      if tl vars = ⟨⟩ then
9        (as1 set, d)
10      else
11       (let (as1 rest, drest) = transform-viewdef(mk-Viewdef0(tl vars))(dict) in
12        if dom d ∩ dom drest ≠ {} then
13          exit("§2.2.2: Two definitions in the same scopeunit use the same name")
14         else
15          (as1 set ∪ as1 rest, d + drest))))
```

type: *Viewdef*$_0$ → *Dict* → *View-definition*$_1$-set *Dict*

Objective Transform a view definition into AS_1

Parameters

vars	The list of (elementary) definitions contained in the (composite) view definition

Result A set of AS_1 view definitions and a *Dict* contribution containing descriptors for the view variables

Algorithm

Line 1	Decompose the first (the next) elementary view definition (the function is recursive).
Line 2	Check that the names in the variable list are unique.
Line 5	Extract the *Qual* of their sort.
Line 6	Extract the *Qual* of the parent sort in the case of a syntype.
Line 7	Transform the first elementary view definition.
Line 8	If there are no more elementary view definitions then return the AS_1 definition and *Dict* contribution for these view variables else
Line 11-15	Join them with the definitions and contributions for the rest of the elementary view definitions provided that the names in the view definition are unique.

transform-view(*varlist*, *tqual*)(*dict*) $\triangleq$ (3.4.10.2)

```
1  (if varlist = ⟨⟩ then
2     ({}, [])
3   else
4    (let mk-Id0(q, nm) = hd varlist in
5     let as1 tid = make-as1-identifier(tqual)(dict),
6         (as1 rest, drest) = transform-view(tl varlist, tqual)(dict) in
7     let qual = (let bqual = get-sur(dict(SCOPEUNIT)) in
8                 if (∃!vqual ∈ dom dict)
9                    ((get-sur(get-sur(vqual)) = bqual) ∧
10                   (∃q')(q' ⌢ q ⌢ ⟨(VALUE, nm)⟩ = vqual) ∧
11                   is-VarD(dict(vqual)) ∧
12                   (let mk-VarD(tq, attr, , ) = dict(vqual) in
13                    tq = tqual ∧ attr = REVEALED)) then
14                  (let vqual be s.t. (get-sur(get-sur(vqual)) = bqual ∧
15                                      is-VarD(dict(vqual)) ∧
16                                      (∃q')(q' ⌢ q ⌢ ⟨(VALUE, nm)⟩ = vqual) ∧
17                                      (let mk-VarD(tq, attr, , ) = dict(vqual) in
18                                       tq = tqual ∧ attr = REVEALED)) in
19                   vqual)
20                 else
21                  exit("§2.6.1.2: No unique corresponding revealed variable of that sort")) in
22     let as1 id = make-as1-identifier(qual)(dict) in
23     let as1 tree = mk-View-definition1(as1 id, as1 tid) in
24     ({as1 tree} ∪ as1 rest, drest + [dict(SCOPEUNIT) ⌢ ⟨(VIEW, qual)⟩ ↦ mk-ViewD(qual)]))
```

type : Id_0^* *Qual* $\rightarrow$ *Dict* $\rightarrow$ *View-definition*$_1$-**set** *Dict*

Objective Transform a list of view variables into AS_1 view definitions

Parameters

varlist	The view variables
tqual	The *Qual* of the variables sort

Result The AS_1 definitions

Algorithm

Line 1	When through, return nothing (the function is recursive).
Line 4	Decompose the first variable identifier.
Line 5	Transform the variable sort into an AS_1 identifier.
Line 6	Transform the rest of the view variables.
Line 7-21	Denote the *Qual* of the corresponding revealed variable by *qual*.
Line 7	Construct the *Qual* of the surrounding block.
Line 8	There must exist a unique variable identifier (i.e. variable *Qual*) which is defined in a process of the same surrounding block,
Line 10	which contains the qualifier and name specified in the view definition,
Line 11	which is a variable,
Line 13	which is of the same sort and which is revealed.
Line 14-19	If there exist such an identifier then let *vqual* denote its *Qual*.
Line 22-23	Construct the AS_1 view definition and
Line 24	Join it with the definitions containing the rest of the view variables (*as*$_1$ *rest*). Also join the view descriptor (*ViewD*) with the descriptors for the rest of the view definitions

3.4.11 Import Definitions

transform-importdef(**mk**-*Importdef*$_0$(*importelemlist*))(*dict*) $\triangleq$ (3.4.11.1)

```
1  (let mk-Importelem0(varl, tid) = hd importelemlist in
2   let tqual = get-visible-qual(tid, TYPE)(dict) in
3   let pqual = get-parent(tqual)(dict) in
4   let scope = dict(SCOPEUNIT) in
5   let delem = [scope ⌢ ⟨(IMPORT, (varl[i], pqual))⟩ ↦ mk-ImportD(tqual) | 1 ≤ i ≤ len varl] in
6   if tl importelemlist = ⟨⟩ then
7     ({}, delem)
8    else
9     (let drest = transform-importdef(tl importelemlist)(dict) in
10     if dom delem ∩ dom drest ≠ {} then
11       exit("§2.2.2: Two definitions in the same scopeunit use the same name")
12      else
13       ({}, delem + drest)))
```

type: *Importdef*$_0$ → *Dict* → *Decl*$_1$-**set** *Dict*

Objective Construct the *Dict* contribution for an imported entity.

Parameters The import definition containing:

importelemlist A list of (elementary) import definitions

Result The *Dict* contributions for the import variables and an empty set (see *transform-signallistdef*).

Algorithm

Line 1 Decompose the first elementary import definition (the function is recursive).

Line 2 Extract the sort *Qual* of the contained variables (*varl*).

Line 3 Extract the parent sort in the case of a syntype.

Line 4-5 Construct the *Dict* contributions for the contained variables. Note that *pqual* is used in the *Qual* in order to achieve uniqueness of the pair of import name- sort, while *tqual* is used in the descriptor since the implicit variable associated with import expressions may be of syntypes.

Line 6-9 And join them with the contributions for the rest of the elementary view definitions (if any).

Line 10 The pairs of import name and sort must be unique

3.4.12 Signalroute Definitions

transform-signalroutedef(mk-*Sigroutedef*$_0$(*nm*, *path*, *opath*))(*dict*) $\triangleq$ (3.4.12.1)

```
 1  (let mk-Sigroutepath0(endpoint1, endpoint2, siglist) = path in
 2   let sigroutequal = dict(SCOPEUNIT) ⌢ ⟨(SIGNALROUTE, nm)⟩ in
 3   let sigset = transform-signallist(siglist)(dict) in
 4   let p1 = get-visible-qual(endpoint1, PROCESS)(dict),
 5       p2 = get-visible-qual(endpoint2, PROCESS)(dict) in
 6   let osigset =
 7       if opath = nil then
 8         {}
 9        else
10         (let mk-Sigroutepath0(orig', dest', osiglist) = opath in
11          let p1' = get-visible-qual(orig', PROCESS)(dict),
12              p2' = get-visible-qual(dest', PROCESS)(dict) in
13          if p1 = p2' ∧ p2 = p1' then
14            transform-signallist(osiglist)(dict)
15           else
16            exit("§2.5.2: The second path must denote reverse direction of the first path")) in
17   if p1 ≠ p2 ∧ is-local(p1)(dict) ∧ is-local(p2)(dict) then
18     (let as1p1 = if p1 = ENV then ENVIRONMENT else make-as1-identifier(p1)(dict),
19          as1p2 = if p2 = ENV then ENVIRONMENT else make-as1-identifier(p2)(dict),
20          as1sigidset = make-as1idset(sigset)(dict),
21          as1sigidset' = make-as1idset(osigset)(dict) in
22      let as1path1 = mk-Signal-route-path1(as1p1, as1p2, as1sigidset),
23          as1path2 = if as1sigidset' = {} then
24                       nil
25                     else
26                       mk-Signal-route-path1(as1p2, as1p1, as1sigidset'),
27          descr = mk-SignalrouteD(p1, p2, sigset, osigset) in
28      let as1tree = mk-Signal-route-definition1(name-to-name1(nm), as1path1, as1path2) in
29      ({as1tree}, [sigroutequal ↦ descr]))
30    else
31     if p1 = p2 then
32       exit("§2.5.2: The endpoints of the signal route are not different")
33      else
34       exit("§2.5.2: The endpoints of the signal route are not locally defined"))
```

type : *Sigroutedef*$_0$ → *Dict* → *Signal-route-definition*$_1$-**set** *Dict*

Objective Transform a signalroute definition into AS_1

Parameters An AS_0 signalroute definition containing:

nm	The name of the signal route
path	The first path
opath	The second optional path

Result A set containing an AS_1 signal route definition and a *Dict* contribution containing the signal route descriptor

Algorithm

Line 1	Decompose the first path.
Line 2	Construct the *Qual* denoting the signal route.
Line 3	Transform the signal list in the first path into a set of signal *Quals*.
Line 4-5	Extract the *Qual* of the two endpoints of the first path (*get-visible-qual* can handle the case where the endpoint is ENV).
Line 6-16	Let *osigset* denote the set of signal *Qual* for the opposite direction (the second path).

Line 6	If the second path is omitted then it is the empty set otherwise
Line 10	Decompose the second path.
Line 11-12	Extract the *Qual* of the two endpoints of the second path.
Line 13	The first endpoint of the first path must be equal to the second endpoint of the second path and the second endpoint of the first path must be equal to the first endpoint of the second path
Line 14	Let *osigset* denote the set of signal *Quals* from the second path.
Line 17	The endpoints of the signal route must be different and the endpoints must be locally defined.
Line 18-19	Construct the AS_1 identifiers (or ENVIRONMENT) of the two endpoints.
Line 20-21	Construct the two AS_1 signal sets of the signals conveyed in the two directions.
Line 22-23	Construct the two AS_1 paths of the directions.
Line 27	Construct the signal route descriptor.
Line 28	Construct the AS_1 signal route definition.
Line 29	Return the AS_1 definition and the *Dict* contribution containing the constructed descriptor.
Line 31-34	For clarity, split the error condition into two **exits**

3.4.13 Signallist Definitions

transform-signallistdef(**mk-***Signallistdef*$_0$(*nm*, *signals*))(*dict*) $\triangleq$ (3.4.13.1)

```
1  (let siglistqual = dict(SCOPEUNIT) ⌢ ⟨(SIGNALLIST, nm)⟩ in
2   let () =
3       transform-signallist(signals)(dict + [siglistqual ↦ mk-ErrorD()]) in
4   ({}, [siglistqual ↦ mk-SignallistD(signals)]))
```

type: *Signallistdef*$_0$ $\rightarrow$ *Dict* $\rightarrow$ *Decl*$_1$-**set** *Dict*

Objective Construct the *Dict* contribution for a signal list definition

Parameters The AS_0 signal list definition containing:

nm	The signal list name
signals	The associated list of signals

Result Beside of the *Dict* contribution, also an empty set of AS_1 definitions is returned (such that all "definition transforming" functions can be treated equally)

Algorithm

Line 1	Construct the *Qual* of the signal list.
Line 3	Transform the list of signals into a signal *Qual* set. However, the result is not used as the function only is applied for checking of recursive definition.
Line 4	Return the *Dict* contribution containing the signal list descriptor

transform-signallist(*sigl*)(*dict*) $\triangleq$ (3.4.13.2)

```
 1  (if sigl = ⟨⟩ then
 2     {}
 3    else
 4     (let sigrest = if tl sigl = ⟨⟩ then {} else transform-signallist(tl sigl)(dict) in
 5      let sigsetelem =
 6          cases hd sigl:
 7           (mk-Id0(,)
 8              → {signal-qual(hd sigl)(dict)},
 9            mk-Signallistid0(id)
10              → (let qual = get-visible-qual(id, SIGNALLIST)(dict) in
11                 let mk-SignallistD(siglist) = dict(qual) in
12                 transform-signallist(siglist)(dict + [qual ↦ mk-ErrorD()]))) in
13      if sigsetelem ∩ sigrest ≠ {} then
14        exit("§2.5.5: Signal identifiers in signal list are not distinct")
15       else
16        sigsetelem ∪ sigrest))
```

type : *Signallist*$_0$ → *Dict* → *Signalqual*-set

Objective Transform a list of signals into a set of *Qual*s

Parameters

sigl	The list of signals

Algorithm

Line 4	Transform all but the first element in the list (the function is recursive).
Line 5-12	Let *sigsetelem* denote the *Qual*s of the first element in the list.
Line 7	If the first element in the list is an identifier then it denotes a signal and *sigsetelem* denotes the set containing that signal *Qual*.
Line 9	If the first element in the list is a signal list then let *qual* denote the *Qual* of the signal list.
Line 11	Let *siglist* denote the AS$_0$ signal list attached to the signal list identifier.
Line 12	Transform that signal list, but invalidate the signal list identifier by denoting it as a *ErrorD* in *Dict*.
Line 13	If any signal *Qual* from the first element also appear in another element then it is an error otherwise
Line 16	The union of the set from the first element and the set from the rest of the list is returned

3.4.14 Connect Statements

transform-block-connect(*decllist*, *sigrouteset*, *connectmap*)(*dict*) $\triangleq$ (3.4.14.1)

```
 1  (if decllist = ⟨⟩ then
 2     (let bqual = dict(SCOPEUNIT) in
 3      let outerchanset = {mk-ChannelD(endp1, endp2, , , newc) ∈ rng dict |
 4                          bqual ∈ {endp1, endp2} ∧ newc = nil} in
 5      if outerchanset ≠ dom connectmap then
 6        exit("§2.5.3: No connection for channel connected to block")
 7       else
 8        if sigrouteset = union rng connectmap then
 9          ({}, connectmap)
10         else
11          exit("§2.5.3: Signal route does not appear in exactly one connect"))
12    else
13     ( cases hd decllist:
14       (mk-Sigroutedef0(nm, mk-Sigroutepath0(end1, end2, ), )
15          → (let qset = if end1 = ENV ∨ end2 = ENV
16                        then {dict(SCOPEUNIT) ⌢ ⟨(SIGNALROUTE, nm)⟩}
17                        else {} in
18             let newset = sigrouteset ∪ qset in
19             transform-block-connect(tl decllist, newset, connectmap)(dict)),
20        mk-Connect0(, )
21          → (let (channel, routeset, connect) = block-connect(hd decllist)(dict) in
22             if channel ∈ dom connectmap then
23               exit("§2.5.3: Channel to signal route connection appears twice")
24              else
25               if union rng connectmap ∩ routeset ≠ {} then
26                 exit("§2.5.3: Signal route mentioned in more than one connection")
27                else
28                 (let nconnectmap = connectmap + [channel ↦ routeset] in
29                  let (restas1, restmap) =
30                      transform-block-connect(tl decllist, sigrouteset, nconnectmap)(dict) in
31                  ({connect} ∪ restas1, restmap))),
32        T → transform-block-connect(tl decllist, sigrouteset, connectmap)(dict))))
```

type: *Decl*$_0$* *Qual*-set *BlockconnectionD* → *Dict* → *Channel-to-route-connection*$_1$-set *BlockconnectionD*

Objective Transform the connect definitions of a block into a set of AS_1 channel to route connections

Parameters

decllist	The AS_1 definition list of a block
sigrouteset	The set of *Quals* denoting the signal routes which already have been handled (recursively). Initially, this set is empty
connectmap	A map from *Quals*, denoting the channels which already have occurred in a connect definition, into the signal route *Quals* to which the channels are connected. Initially, this map is empty

Result The constructed AS_1 connect definitions

Algorithm

Line 2	Let *bqual* denote the block
Line 3	Let *outerchanset* denote the channels having the block as one of their endpoints. Do not include the channels which have been replaced by two new channels (i.e. *newc* = **nil**).
Line 5-6	The set of channels having the block as one of its endpoints must be equal to the set of channels mentioned in a connection in the block.

Line 8-11	Return if the set of signal route *Qual*s derived from the signal route definitions in the block is the same as the constructed set, by merging the sets contained in the *connectmap* i.e. all signal routes mentioned in a connect must be defined in the block and having the environment as one of the endpoints and vice versa.
Line 14-19	If the first (the next) definition in the list is a signal route definition then include the signal *Qual* (contained in *qset*) to *sigrouteset* provided that one of the endpoints are ENV.
Line 20	If the next definition is a block connect then
Line 21	Derive the channel *Qual*, the signal route *Qual*s and the AS_1 connect definition from the block connect.
Line 22	If the channel already has been used in a block connect then it is an error.
Line 25	Each signal route must be mentioned in only one channel to signal route connection.
Line 28	Update the connectmap to include the connect information in hand
Line 32	If the next definition is anything else then continue with the rest of the definitions

block-connect(**mk-***Connect*$_0$(*chid*, *routeidlist*))(*dict*) $\triangleq$ (3.4.14.2)

```
1  (let cqual = get-visible-qual(chid, CHANNEL)(dict) in
2   let mk-ChannelD(, endpoint2, sigset1, sigset2, newc) = dict(cqual) in
3   let bqual = dict(SCOPEUNIT) in
4   let connectset = {get-visible-qual(routeidlist[i], SIGNALROUTE)(dict) | i ∈ ind routeidlist} in
5   let (insigset, outsigset) = if endpoint2 = bqual then (sigset1, sigset2) else (sigset2, sigset1) in
6   let cas1id = make-as1-identifier(cqual)(dict),
7       sas1set = {make-as1-identifier(cq)(dict) | cq ∈ connectset} in
8   let cqual' = if newc = nil then cqual else convert-channel-qual(newc, bqual) in
9   if card connectset = len routeidlist ∧
10    is-wf-connectsignalroutes(connectset, insigset, outsigset, {}, {})(dict) then
11    (cqual', connectset, mk-Channel-to-route-connection1(cas1id, sas1set))
12    else
13    exit("§2.5.3: Channel to signal route connection is not well-formed"))
```

type: *Connect*$_0$ → *Dict* → *Channelqual Qual*-**set** *Channel-to-route-connection*$_1$

Objective Derive the channel *Qual*, the set of signal route *Qual*s and an AS_1 connect definition from an AS_0 channel to route connection

Parameters

chid	The AS_0 channel identifier in the connect
routeidlist	The AS_0 list of signal route identifiers
Result	The *Qual* of the channel in the enclosing scopeunit, a *Qual* set containing the signal routes connected to that channel and the AS_1 connection.

Algorithm

Line 1	Extract the *Qual* of the channel.
Line 2	Decompose the channel descriptor.
Line 3	Let *bqual* denote the *Qual* of the block containing the connect definitions.
Line 4	Construct the set of *Qual*s denoting the signal route identifiers contained in the channel to route connection.
Line 5	Let *insigset* denote the incoming signals and let *outsigset* denote the outgoing signals.

Line 6-7	Construct the AS_1 identifier of the channel and the AS_1 identifier set containing the signal routes.
Line 8	If the channel has a substructure then use the *Qual* of the appropriate replacing channel.
Line 9	A signal route identifier must not be mentioned twice in the connect definition and
Line 10	The signals in the signal routes must be mentioned in the channel.
Line 11	Return the *Qual* of the channel, the set of signal route *Qual*s and an AS_1 channel to route connection

is-wf-connectsignalroutes(*connectset*, *insigset*, *outsigset*, *res*1, *res*2)(*dict*) $\triangleq$ (3.4.14.3)

```
1  if connectset = {} then
2    insigset = res1 ∧ outsigset = res2
3   else
4    (let squal ∈ connectset in
5     let mk-SignalrouteD(endpoint1, endpoint2, sset1, sset2) = dict(squal) in
6     if ENV ∉ {endpoint1, endpoint2} then
7       false
8       else
9       (let (inset, outset) = if endpoint1 = ENV then (sset1, sset2) else (sset2, sset1) in
10       is-wf-connectsignalroutes(connectset \ {squal}, insigset, outsigset,
11                                 inset ∪ res1, outset ∪ res2)(dict)))
```

type : *Qual*-**set** *Signalqual*-**set** *Signalqual*-**set** *Signalqual*-**set** → *Dict* → *Bool*

Objective Check that signal routes in a channel to route connection are properly interfaced

Parameters

connectset	The (remaining) set of signal route *Qual*s (the function is recursive)
insigset	The set of signal *Qual*s leading to the block via the channel specified in the channel to route connection
outsigset	The set of signal *Qual*s leading out of the block via the channel specified in the channel to route connection
res1,*res2*	The recursively created sets of input signals and output signals specified in the signal routes. Eventually, these set must be equal to the corresponding set specified for the channel

Result The collected set of signal of incoming signal (*res1*) and the outgoing signals (*res2*) must be equal to the incoming respective outgoing signals specified in the channel

Algorithm

Line 1	When through, return true if all signals from the channel have been mentioned in a signal route.
Line 4	Let *squal* denote a *Qual* in the signal route set.
Line 5	Decompose the *Dict* descriptor for the signal route
Line 6	ENV must be one of the endpoints for the signal route.
Line 9	Let *inset* denote the signals, taken from the signal route descriptor, which leads into the block and let *outset* denote the signals which leads out of the block.
Line 10	The rest of the signal routes in the connect must be properly interfaced

transform-substructure-connect(*decllist*, *channelset*, *connectmap*)(*dict*) ≙ **(3.4.14.4)**

```
 1  (if decllist = ⟨⟩ then
 2     (let bqual = get-sur(dict(SCOPEUNIT)) in
 3      let outerchanset = {mk-ChannelD(endp1, endp2, , , newc) ∈ rng dict |
 4                          bqual ∈ {endp1, endp2} ∧ newc = nil} in
 5      if card outerchanset > card dom connectmap then
 6        exit("§3.2.2: No connection for channel connected to block substructure")
 7       else
 8        if channelset = union rng connectmap then
 9          {}
10         else
11          exit("§3.2.2: Sub-channel does not appear in exactly one connect"))
12    else
13     ( cases hd decllist:
14       (mk-Chandef0(nm, mk-Chanpath0(orig, dest, ), , , )
15          → (let qset = if orig = ENV ∨ dest = ENV
16                        then {dict(SCOPEUNIT) ⌢ ⟨(CHANNEL, nm)⟩}
17                        else {} in
18             let newset = channelset ∪ qset in
19             transform-substructure-connect(tl decllist, newset, connectmap)(dict)),
20        mk-Connect0(, )
21          → (let (chan, chanset, connect) = transform-block-substructure-connect(hd decllist)(dict) in
22             if chan ∈ dom connectmap then
23               exit("§3.2.2: Channel connection appears twice")
24              else
25               if union rng connectmap ∩ chanset ≠ {} then
26                 exit("§3.2.2: Sub-channel mentioned in more than one connection")
27                else
28                 (let nconnectmap = connectmap + [chan ↦ chanset] in
29                  {connect} ∪
30                  transform-substructure-connect(tl decllist, channelset, nconnectmap)(dict))),
31        T → transform-substructure-connect(tl decllist, channelset, connectmap)(dict))))
```

type : $Decl_0^*$ *Channelqual*-**set** (*Channelqual* $\overrightarrow{m}$ *Channelqual*-**set**) → *Dict* → *Channel-connection*$_1$-**set**

Objective Construct the channel connections attached to the block sub-structure

Parameters

decllist — The AS_0 definition list of the block sub-structure joined with the AS_0 connections which connects the implicit export - import channel.

channelset, *connectmap*, — During the recursive traverse through the definition list *decllist*, channel names from channel definition leading from or to the environment are collected and kept in *channelset*, connect information is collected and kept in *connectmap*. When *transform-substructure-connect* initially is applied (in *transform-blockdef*), these parameters are empty.

Result The collected AS_1 connections.

Algorithm

Line 2 — Let *bqual* denote the block surrounding the block substructure

Line 3 — Let *outerchanset* denote the channels having the block as one of their endpoints and which do not have a sub-structure.

Line 5-6 — If there are more channels having the block as one of their endpoints than channels in channel connections then some channel connections are missing.

Line 8-11	If all channels (having one of the endpoints as the environment) have occurred in a connection then return.
Line 14	If the definition in hand is a channel definition then
Line 15-18	Construct the set consisting of the empty set if the channel do not lead from or to the environment, otherwise consisting of the channel *Qual*.
Line 19	Traverse through the rest of the definition list with the set added to *channelset*
Line 21	If the definition in hand is a channel connection then transform the connection into AS_1, obtaining the channel *Qual* (*chan*), the set of channel *Quals* (*chanset*), and the AS_1 connection *connect*.
Line 22	The channel must not be contained in more than one channel connection.
Line 25	A sub-channel must be mentioned in only one channel connection.
Line 28-29	Traverse through the rest of the definition list with the connect information added to *connectmap*.

transform-block-substructure-connect(mk-*Connect*$_0$(*chid*, *chidlist*))(*dict*) $\triangleq$ **(3.4.14.5)**

```
 1  (let cqual = get-visible-qual(chid, CHANNEL)(dict) in
 2   let mk-ChannelD(endpoint1, endpoint2, sigset1, sigset2, newc) = dict(cqual) in
 3   let bqual = get-sur(dict(SCOPEUNIT)) in
 4   if bqual ∈ {endpoint1, endpoint2} then
 5     (let connectset = {get-visible-qual(chidlist[i], CHANNEL)(dict) | i ∈ ind chidlist} in
 6      let (insigset, outsigset) = if endpoint2 = bqual then
 7                                    (sigset1, sigset2)
 8                                  else
 9                                    (sigset2, sigset1) in
10      let cas1id = make-as1-identifier(cqual)(dict),
11          cas1list = {make-as1-identifier(cq)(dict) | cq ∈ connectset} in
12      let cqual' = if newc = nil then cqual else convert-channel-qual(newc, bqual) in
13      if card connectset = len chidlist ∧
14        is-wf-connectchannels(connectset, insigset, outsigset, {}, {})(dict) then
15        (cqual', connectset, mk-Channel-connection1(cas1id, cas1list))
16        else
17         exit("§3.2.2: Channel connection is not well-formed"))
18    else
19     exit("§3.2.2: Channel connection is not well-formed"))
```

type: *Connect*$_0$ → *Dict* → *Channelqual* *Channelqual*-**set** *Channel-connection*$_1$

Objective Transform a channel connect into AS_1

Parameters The channel connection containing:

chid	The identifier of the channel in the enclosing scopeunit
chidlist	The list of channels connected to *chid*

Result The *Qual* of the channel in the enclosing scopeunit, a *Qual* set containing the channels connected to that channel and the AS_1 channel connection

Algorithm

Line 1	Construct the *Qual* of *chid*.
Line 2	Decompose its channel descriptor.
Line 3	Let *bqual* denote the *Qual* of the enclosing block.
Line 4	The block must be one of the endpoints of the channel mentioned in the connect definition.

Line 5	Construct a set containing the *Quals* of the channels connected to *chid.*
Line 6	Let *insigset* denote the signal *Qual* set leading into the sub-structure via the channel and let *outsigset* denote the signal *Qual* set leading out of the sub-structure via the channel.
Line 10-11	Construct the AS_1 identifier of the outer channel and the set of AS_1 identifiers of the channels to which it is connected.
Line 12	If the channel has a substructure then use the *Qual* of the appropriate replacing channel.
Line 13	A channel must not be mentioned twice in a channel connection and
Line 14	The signals in the channels must be properly interfaced to the outer channel.
Line 15	If everything is OK then return the *Qual* of the outer channel, the *Qual* set of the channels to which it is connected and an AS_1 channel connection

is-wf-connectchannels(*connectset*, *insigset*, *outsigset*, *res*1, *res*2)(*dict*) $\triangleq$ (3.4.14.6)

```
1  (if connectset = {} then
2     is-wf-refinement(insigset, outsigset, res1, res2)(dict)
3   else
4    (let cqual ∈ connectset in
5     let mk-ChannelD(endpoint1, endpoint2, sset1, sset2, ) = dict(cqual) in
6     if ENV ∉ {endpoint1, endpoint2} then
7       false
8      else
9       (let (inset, outset) = if endpoint1 = ENV then (sset1, sset2) else (sset2, sset1) in
10       is-wf-connectchannels(connectset \ {cqual}, insigset, outsigset, res1 ∪ inset, res2 ∪ outset)(dict))))
```

type : *Channelqual*-**set** *Signalqual*-**set** *Signalqual*-**set** *Signalqual*-**set** *Signalqual*-**set** → *Dict* → *Bool*

Objective Check that channels in a channel connection are properly interfaced

Parameters

connectset	The (remaining) set of channel *Quals* (the function is recursive)
insigset	The set of signal *Quals* leading to the block sub-structure via the sub-channel
outsigset	The set of signal *Quals* leading out of the block sub-structure via the sub-channel
res1,res2	The recursively created sets of input signals and output signals specified in the sub-channels.

Result True if success

Algorithm

Line 1	When through, check that the constructed signal sets (*res1,res2*) are well-formed refinements of the signals specified for the outer channel.
Line 4	Let *squal* denote a *Qual* in the sub-channel set.
Line 5	Decompose the *Dict* descriptor for the sub-channel
Line 6	ENV must be one of the endpoints for the sub-channel.
Line 9	Let *inset* denote the signals, taken from the sub-channel descriptor, which leads into the block sub-structure and let *outset* denote the signals which leads out of the block sub-structure.
Line 9	Return **true** if the rest of the sub-channels in the connect is properly interfaced

is-wf-refinement(*parentset1*, *parentset2*, *subset1*, *subset2*)(*dict*) $\triangleq$ (3.4.14.7)

```
1  (if (∃q ∈ subset1 ∪ subset2)(get-sur(q) ∈ parentset1) then
2     (let q ∈ subset1 ∪ subset2 be s.t. get-sur(q) ∈ parentset1 in
3      let parent = get-sur(q) in
4      let mk-SignalD(, sub1, sub2) = dict(parent) in
5      if q ∈ subset1 ∧ sub1 ⊆ subset1 ∧ sub2 ⊆ subset2 then
6        is-wf-refinement(parentset1 \ {parent}, parentset2, subset1 \ sub1, subset2 \ sub2)(dict)
7       else
8        false)
9    else
10    (if (∃q ∈ subset2 ∪ subset1)(get-sur(q) ∈ parentset2) then
11       (let q ∈ subset1 ∪ subset2 be s.t. get-sur(q) ∈ parentset2 in
12        let parent = get-sur(q) in
13        let mk-SignalD(, sub1, sub2) = dict(parent) in
14        if q ∈ subset2 ∧ sub1 ⊆ subset2 ∧ sub2 ⊆ subset1 then
15          is-wf-refinement(parentset1, parentset2 \ {parent}, subset1 \ sub2, subset2 \ sub1)(dict)
16         else
17          false)
18       else
19        parentset1 = subset1 ∧ parentset2 = subset2))
```

type: *Signalqual*-set *Signalqual*-set *Signalqual*-set *Signalqual*-set → *Dict* → *Bool*

Objective Check that the signal refinement at a sub-structure boundary is made properly

Parameters

parentset1	The set of incoming signal *Quals* conveyed by the outer channel
parentset2	The set of outgoing signal *Quals* conveyed by the outer channel
subset1	The set of incoming signal *Quals* conveyed by the sub-channels
subset2	The set of outgoing signal *Quals* conveyed by the sub-channels

Result True if success

Algorithm

Line 1	If there exist a signal conveyed by a sub-channel which has a incoming parent signal conveyed by the outer channel then refinement of an incoming signal has been used.
Line 2	Extract any sub-signal.
Line 3	Extract the parent of the sub-signal.
Line 4	Decompose the signal descriptor in order to access its sub-signals *sub1* and *sub2*.
Line 5	As one of the sub-signals is specified in the sub-channels, all of the sub-signals must be specified, that is the sub-signals in the two directions (*sub1, sub2*) must be subsets of the signals specified for the sub-channels.
Line 6	Remove the parent signal and its sub-signals from the parameters before checking for further refinements.
Line 10-17	Do the same as in line 1-6, but where the presence of an outgoing parent signal is investigated.
Line 19	If there is no (more) refinement then the incoming signals conveyed by the outer channel must be equal to the incoming signals conveyed by the sub-channels and the outgoing signals conveyed by the outer channel must be equal to the outgoing signals conveyed by the sub-channels

service-connectmap(*decllist*, *sigrouteset*, *connectmap*)(*dict*) $\triangleq$ (3.4.14.8)

```
1  (if decllist = ⟨⟩ then
2     (let pqual = dict(SCOPEUNIT) in
3      let outersigrouteset = {mk-SignalrouteD(endp1, endp2, , ) ∈ rng dict | pqual ∈ {endp1, endp2}} in
4      if sigrouteset = {} ∧ connectmap = [] then
5        []
6       else
7        if card outersigrouteset > card dom connectmap
8          then exit("§4.10.2: Missing signal route connection in process")
9          else if sigrouteset ≠ union rng connectmap
10               then exit("§4.10.2: All service signal routes must be mentioned in a connect")
11               else connectmap)
12    else
13    ( cases hd decllist:
14      (mk-Sigroutedef0(nm, mk-Sigroutepath0(end1, end2, ), )
15        → (let qset = if end1 = ENV ∨ end2 = ENV
16                      then {dict(SCOPEUNIT) ⌢ ⟨(SIGNALROUTE, nm)⟩}
17                      else {} in
18           let newset = sigrouteset ∪ qset in
19           service-connectmap(tl decllist, newset, connectmap)(dict)),
20       mk-Connect0(, )
21         → (let (route, routeset) = service-connect(hd decllist)(dict) in
22            let nconnectmap = connectmap + [route ↦ routeset] in
23            if route ∉ dom connectmap ∧ routeset ∩ union rng connectmap = {} then
24              service-connectmap(tl decllist, sigrouteset, nconnectmap)(dict)
25             else
26              exit("§4.10.2: Illegal service signal route connection")),
27       T → service-connectmap(tl decllist, sigrouteset, connectmap)(dict))))
```

type : *Decompositiondecl*$_0^*$ *Qual*-**set** *ProcessconnectionD* → *Dict* → *ProcessconnectionD*

Objective Check that the service signal routes of a service decomposition are properly connected

Parameters

decllist	The definition list of a decomposition
sigrouteset	The *Quals* of the service routes which have ENV specified as one of the endpoints. The service signal routes are collected during the recursive traverse through the definition list. When the function initially is applied (in *transform-decomposition-body*) the set is empty
connectmap	A map which defines the relationship between the signal routes and the service signal routes to which it is connected. This map is constructed during the recursive traverse through the definition list

Result The above mentioned *ProcessconnectionD*. It is used for checking of VIA constructs in services.

Algorithm

Line 1-9	When all the connections and service signal routes have been considered it must hold that the set of service signal routes equals the service signal routes mentioned in a service signal route connect (line 9).
Line 2	Let *pqual* denote the process
Line 3	Let *outersigrouteset* denote the signal routes having the process as one of their endpoints

Line 4	If no service signal routes (and connections) are specified for the process then return the empty map.
Line 7	If there are more signal routes having the process as one of their endpoints than signal routes in service signal route connections then some service signal route connections are missing (line 7).
Line 13	Consider the next definition:.
Line 14-18	If the next definition is a service signal route definition and one of the endpoints is ENV then let *newset* denote the set which is updated to include that service signal route.
Line 19	Continue with the next definition.
Line 20	If the next definition is a service signal route connection then
Line 21	Check that the connection is well-formed and derive the *Qual* of the signal route (*route*) and the *Quals* of the service signal routes to which it is connected (*routeset*).
Line 22	Update the connection map to include information about this connection
Line 23	No signal route or service signal route may be mentioned in more than one connection. route and the signal routes in this connection must not be connected in another connection.
Line 24	Continue with the rest of the definitions.
Line 27	Continue with the rest of the definitions

service-connect(**mk-***Connect*$_0$(*sigrouteid*, *routeidlist*))(*dict*) $\triangleq$ (3.4.14.9)

```
1  (let squal = get-visible-qual(sigrouteid, SIGNALROUTE)(dict) in
2   let mk-SignalrouteD(, endpoint2, sigset1, sigset2) = dict(squal) in
3   let pqual = dict(SCOPEUNIT) in
4   let connectset = {get-visible-qual(routeidlist[i], SIGNALROUTE)(dict) | i ∈ ind routeidlist} in
5   let (insigset, outsigset) = if endpoint2 = pqual then (sigset1, sigset2) else (sigset2, sigset1) in
6   if card connectset ≠ len routeidlist then
7     exit("§4.10.1: Service signal route identifier occurs twice in service signal route connection")
8    else
9     if is-wf-connectsignalroutes(connectset, insigset, outsigset, {}, {})(dict) then
10      (squal, connectset)
11     else
12      exit("§4.10.1: Signals in service signal routes must be the same as for the connected signal route"))
```

type: *Connect*$_0$ → *Dict* → *Qual Qual*-set

Objective Check that a service signal route connection is well-formed

Parameters The AS$_0$ service signal route connection containing

sigrouteid	The signal route
routeidlist	The service signal routes

Result The *Qual* of the signal route and the set of service signal route *Quals*

Algorithm

Line 1	Construct the *Qual* of the signal route.
Line 2	Decompose the signal route descriptor.
Line 3	Let *pqual* denote the *Qual* of the surrounding process
Line 4	Construct the *Quals* of the service signal routes.
Line 5	Let *insigset* denote signals in the signal route which leads to the process and let *outsigset* denote the signals which leads out of the signal route.
Line 6-7	The service signal routes in the connection must be disjoint and

Line 9 The signals in the signal route must match the signals in the service signal routes

3.5 Transformation of Expressions

This section contains the functions which transform expressions and terms into AS_1 expressions and terms. They also takes care of the "resolution by context" problem, that is, sometimes the expression transforming functions are called for the sake of identifying the sort of an expression. Therefore, the functions also takes as argument a set of allowed sorts and they return a set of allowed sorts which always is a non-empty subset of the argument set. For instance, in the transformation of equations, the set of all visible sorts are initially given as argument when the left-hand side is transformed. The resulting set of sorts is then used when the right-hand side is transformed. The resulting set of sorts of this transformation must then contain exactly one sort. The left-hand side and the right-hand side is then transformed again, this time with that particular sort given as argument.

In addition, information about the *Context* in which the expression/term is used is given as argument. Four different kind of contexts are used: CONSTANT denoting that the argument expression/term must be a constant expression, EXPRESSION denoting that the argument may be a non-constant expression, AXIOMS denoting that the argument must be a term used in the axioms and MAPPING denoting that the argument must be a term used in the mapping part of the axioms.

During the transformation, information is collected about the import expressions and value identifiers occurring in the expression/term. Therefore also a *Dict* is returned which may contain value identifier descriptors (*ValueidD*) and may contain information about the import expressions in the *Quotdict* entries IMPLIED and IMPORTLIST (see the domain definition of *Quotdict*).

The entry functions are the two (identical) functions *transform-term* and *transform-expr*

transform-term(*term*, *context*, *sortset*)(*dict*) $\triangleq$ **(3.5.1)**

1 *transform-expr*(*term*, *context*, *sortset*)(*dict*)

type : $Term_0$ *Context Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ $Term_1$ *Sortqual*-**set** *Dict*

Objective Transform an AS_0 term into an AS_1 term. The function is identical to *transform-expr*. It is introduced for the sake af readability.

transform-expr(*expr*, *context*, *sortset*)(*dict*) $\triangleq$ (3.5.2)

```
1  (let (as1tree, sortset', dict') =
2      ( cases expr:
3        (mk-Scopeexpr0(scope, ex)
4           → transform-expr(ex, context, sortset)(dict + [SCOPEUNIT ↦ scope]),
5         mk-Id0(,)
6           → if context ∈ {AXIOMS, MAPPING} then
7                transform-axiom-id(expr, sortset)(dict)
8              else
9                transform-id(expr, context, sortset)(dict),
10        mk-Stringterm0(,)
11          → transform-stringexpr(expr, sortset)(dict),
12        mk-Condexpr0(e1, e2, e3),
13        mk-Condterm0(e1, e2, e3)
14          → transform-condexpr(mk-Condexpr0(e1, e2, e3), context, sortset)(dict),
15        mk-Operatorapp0(,)
16          → transform-operatorexpr(expr, context, sortset)(dict),
17        mk-Operatorterm0(,)
18          → transform-operatorterm(expr, context, sortset)(dict),
19        mk-Monadexpr0(op, ex),
20        mk-Monadterm0(op, ex)
21          → transform-monadexpr(mk-Monadexpr0(op, ex), context, sortset)(dict),
22        mk-Infixexpr0(e1, op, e2),
23        mk-Infixterm0(e1, op, e2)
24          → transform-infixexpr(mk-Infixexpr0(e1, op, e2), context, sortset)(dict),
25        mk-Errorterm0()
26          → exit("§5.4.1.7: Error term is used in a composite term"),
27        mk-Spellingterm0()
28          → transform-spelling(expr, context, sortset)(dict),
29        T → transform-build-in-expression(expr, context, sortset)(dict))) in
30 let as1tree' = if context ∉ {AXIOMS, MAPPING} ∧ is-Ground-term1(as1tree) then
31                    mk-Ground-expression1(as1tree)
32                  else
33                    as1tree in
34 (as1tree', sortset', dict'))
```

type: (*Expr*$_0$ | *Term*$_0$ | *Scopeexpr*$_0$) *Context Sortqual*-**set** → *Dict* →
[*Expression*$_1$ | *Term*$_1$] *Sortqual*-**set** *Dict*

Objective Transform an AS_0 expression into an AS_1 expression

Parameters

expr	The AS_0 expression or term
context	The (syntactic) context in which *expr* is used (see above).
sortset	The set of legal result sorts for the expression/term.

Result

- The resulting AS_1 expression or term.
- The resulting set of (possible) legal result sorts of the expression. This set will always be a subset of *sortset*. If the cardinality of the set is greater than one then the resulting AS_1 expression/term is not used (= **nil**) due to ambiguity. (Which not necessarily is an error as *transform-expr* also takes care of "resolution by context" see above).
- *dict* is updated with information about possible implicit value identifiers contained in a term and with implicit import variables (see above).

Algorithm

Line 3	If the expression is the special synthetic $Scopeexpr_0$ (see the definition of AS_0) then transform the contained expression in the context of the service denoted by *scope*.
Line 5-9	If the expression is an identifier then transform it as a literal or a value identifier if the expression occurs in the axioms else as a literal, a synonym or a variable.
Line 10	Transform a character string.
Line 12-13	Transform a conditional expression or a conditional term.
Line 15	Transform an operator application applied in expression context.
Line 17	Transform an operator application applied in term context.
Line 19-20	Transform a monadic expression or term.
Line 22-23	Transform a dyadic expression or term.
Line 25	The error term may generally not occur in terms (or expressions). The special cases where it is allowed are excluded before applying *transform-term*.
Line 27	Transform the spelling term.
Line 29	Transform the other (imperative) kind of expressions.
Line 30-33	If the transformed expression/term is constant then it is a $Ground\text{-}expression_1$.
Line 34	Return the AS_1 term/expression, the restricted set of allowed sorts, and the *Dict* possibly containing value identifier descriptors and information about contained import expressions

3.5.1 Identifiers

transform-axiom-id(*id*, *sortset*)(*dict*) $\triangleq$ (3.5.1.1)

```
 1  if card sortset = 1 then
 2    transform-axiom-id-of-this-sort(id, sortset)(dict)
 3   else
 4    (let mk-Id0(q, nm) = id in
 5     let litset = all-visible-literals(id, sortset)(dict) in
 6     let litsortset = {get-sur(q) | q ∈ litset} in
 7     let qual = dict(SCOPEUNIT) ⌢ ⟨(VALUE, nm)⟩ in
 8     let sorts = if q ≠ ⟨⟩ then
 9                   {}
10                 else
11                  if qual ∈ dom dict
12                    then (let mk-ValueidD(, so, ) = dict(qual) in
13                            so)
14                    else sortset in
15     let valset = sortset ∩ sorts in
16     let totset = litsortset ∪ valset in
17     if card totset = 0 then
18       exit("§2.2.2: No sort can be used for value identifier")
19      else
20       if card totset = 1 then
21         transform-aziom-id-of-this-sort(id, totset)(dict)
22        else
23         if q = ⟨⟩ then
24           (let mk-ValueidD(val, so, e) = if qual ∈ dom dict then
25                                            dict(qual)
26                                          else
27                                            mk-ValueidD(nil, sortset, false) in
28             if so ∩ sortset = {} then
29               (nil, totset, dict)
30             else
31              (let dict' = dict + [qual ↦ mk-ValueidD(val, so ∩ sortset, e)] in
32               (nil, totset, dict')))
33          else
34           (nil, totset, dict))
```

type: Id_0 *Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ [$Term_1$] *Sortqual*-**set** *Dict*

Objective Transform a term which is a literal or a value identifier into AS_1.

Parameters See *transform-expr* and the introduction to this section

Result See *transform-expr* and the introduction to this section

Algorithm

Line 1 If the sort of the identifier has been determined then transform the identifier given that sort.

Line 4 Decompose the identifier.

Line 5 Construct a *Qual* set of all visible literals having the same AS_0 identifier and being of an allowed sort in this context.

Line 6 Construct a *Qual* set containing the sort of the literals.

Line 7 Construct a value identifier *Qual*.

Line 8-14 If the identifier is qualified then it cannot be a value identifier (the set of possible sorts is empty) else if there exist a descriptor for a value identifier in *Dict* (implied by explicit quantification or by implicit quantification elsewhere) then the set of possible sorts is found in the descriptor (*so*) else the value identifier can be of any sort allowed in this context (*sortset*).

Line 15	Let *valset* denote the set of allowed sorts in this context if the identifier is a value identifier.
Line 16	Let *totset* denote the set of allowed sorts in this context if the identifier denotes a value identifier or a literal.
Line 17	There must exist at least one possible sort.
Line 20-34	If there exist exactly one possible sort then transform the identifier given that sort else return no AS_1 object (**nil**) and return the complete set of allowed sorts. But first, create or modify the value identifier descriptor if the identifier can denote a value identifier (if it is not qualified).
Line 24-27	If a value identifier descriptor does not already exist then create a descriptor.
Line 28-29	If the sort set in the descriptor has no elements in common with the allowed set of sorts in this context then the identifier cannot denote a value identifier and no descriptor is therefore added to *Dict.*
Line 31-32	If the descriptor has elements in common with the allowed set of sorts then restrict the sort set in the descriptor to contain only the common elements. Return the modified *Dict*

transform-id(*id*, *context*, *sortset*)(*dict*) $\triangleq$ (3.5.1.2)

```
1  (let squal = get-visible-qual(id, VALUE)(dict) in
2   let sort = get-parent(s-Sortqual(dict(squal)))(dict) in
3   if is-local(squal)(dict) then
4     if (context = EXPRESSION ∨ is-SynD(dict(squal))) ∧ sort ∈ sortset then
5       if is-SynD(dict(squal)) then
6         transform-expr(s-Expr0(dict(squal)), CONSTANT, {sort})(dict)
7        else
8         (make-as1-identifier(squal)(dict), {sort}, dict)
9      else
10      exit("§2.2.2: Local synonym or variable is not of right sort")
11    else
12     (let totalset = all-visible-literals(id, sortset)(dict) ∪
13          all-variables-and-synonyms(id, context, sortset)(dict) in
14      let squalset = {q | (∃d ∈ totalset)(q = get-parent(s-Sortqual(dict(d)))(dict))} in
15      if totalset = {} then
16        exit("§2.2.2: No synonym, variable or literal matches the sort of the context")
17       else
18        if card squalset = 1 then
19          if card totalset ≠ 1 then
20            exit("§2.2.2: Several variables and synonyms are possible for the context")
21           else
22            (let qual ∈ totalset in
23             let squal ∈ squalset in
24             if is-SynD(dict(qual)) then
25               (let deflevel = [SCOPEUNIT ↦ get-sur(qual)] in
26                transform-expr(s-Expr0(dict(qual)), CONSTANT, {squal})(dict + deflevel))
27              else
28               if is-LiteralD(dict(qual)) then
29                 (mk-Ground-term1(make-as1-identifier(qual)(dict)), {squal}, dict)
30                else
31                 (make-as1-identifier(qual)(dict), {squal}, dict))
32         else
33          (nil, squalset, dict)))
```

type: Id_0 *Context Sortqual*-set $\rightarrow$ *Dict* $\rightarrow$ [$Expression_1$] *Sortqual*-set *Dict*

Objective Transform a literal or a variable or a synonym into AS_1

Parameters See *transform-expr* and the introduction to this section

Result See *transform-expr* and the introduction to this section

Algorithm

Line 1 First use the normal visibility algorithm to find a variable or synonym.

Line 3 If it is defined locally then

Line 4 If the context is constant expression then it must be a synonym and the sort of the variable or synonym must be of a right sort

Line 5 If the identifier denotes a synonym then transform the expression of the synonym else return the AS_1 identifier of the variable and return the sort of the variable.

Line 12-26 If the synonym or variable is not locally defined the place of definition is solved by context.

Line 12-14 The total set of possible *Quals* for visible literals, variables and synonyms is denoted by *totalset* and their sorts are denoted by *squalset*.

Line 15-16 There must exist at least one appropriate variable or synonym or literal.

Line 18-19	If there exist exactly one possible sort for the context then there must also exist exactly one possible identifier and if so then
Line 22	Let *qual* denote the *Qual* of the identifier.
Line 23	Let *squal* denote the *Qual* of the sort.
Line 24-26	If the identifier denotes a synonym then transform the expression of the synonym in the context where the synonym is defined (i.e. *deflevel*).
Line 28-31	Return the AS_1 identifier of the variable or synonym and if the identifier denotes a literal then enclose the identifier in a *Ground-term*$_1$.
Line 33	If there is more than one possible sort then the expression is (still) not unique and **nil** is returned

all-variables-and-synonyms(**mk-***Id*$_0$(*qu*, *nm*), *context*, *sortset*)(*dict*) $\triangleq$ (3.5.1.3)

```
1  (let level = dict(SCOPEUNIT) in
2   {q ∈ dom dict | (is-SynD(dict(q)) ∨ (is-VarD(dict(q)) ∧ context ≠ CONSTANT)) ∧
3                   (∃q')(get-parent(q' ⌢ qu)(dict) ⌢ ⟨(VALUE, nm)⟩ = q ∧
4                         (∃q'')(q ⌢ q'' = level) ∧
5                         get-parent(s-Sortqual(dict(q)))(dict) ∈ sortset)})
```

type: *Id*$_0$ *Context Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ *Qual*-**set**

Objective Extract the *Quals* of all the synonyms and variables which can be used on a given condition

Parameters

Id$_0$	An identifier which qualifies (restricts) the resulting *Qual* set
context	if context equals CONSTANT then only the synonyms are extracted
sortset	A set of sorts which also qualifies the resulting *Qual* set

Algorithm

Line 1	Let *level* denote the enclosing scopeunit.
Line 2-5	Return all those *Quals* which denotes synonyms or variables, includes the qualifier (line 3), is visible (line 4) and which is of the right sort (line 5).

transform-axiom-id-of-this-sort(*id*, *totset*)(*dict*) $\triangleq$ (3.5.1.4)

```
1  (let mk-Id0(qual, nm) = id in
2   let q = dict(SCOPEUNIT) ⌢ ⟨(VALUE, nm)⟩ in
3   if qual = ⟨⟩ ∧ q ∈ dom dict ∧
4     (let mk-ValueidD(, so, e) = dict(q) in
5      so = totset ∧ e) then
6      (let mk-ValueidD(val, , ) = dict(q) in
7       if val = nil then
8         (mk-Composite-term1(make-as1-identifier(q)(dict)), totset, dict)
9        else
10        transform-axiom-id-of-this-sort(as0-id(val), totset)(dict))
11    else
12     (let litset = all-visible-literals(id, totset)(dict) in
13      if litset = {} then
14        if q ≠ ⟨⟩ then
15          exit("§5.2.3: Value identifier must not be qualified")
16         else
17          (let vald = if q ∈ dom dict then
18                        (let mk-ValueidD(, so, ) = dict(q) in
19                         if so ∩ totset = {} then
20                           exit("§5.2.3: Inconsistent use of implicitly quantified value name")
21                          else
22                           mk-ValueidD(nil, so ∩ totset, false))
23                       else
24                        mk-ValueidD(nil, totset, false) in
25           let d = dict + [q ↦ vald] in
26           (mk-Composite-term1(make-as1-identifier(q)(dict)), totset, d))
27       else
28        (let litqual ∈ litset in
29         (mk-Ground-term1(make-as1-identifier(litqual)(dict)), totset, dict))))
```

type: *Id*$_0$ *Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ *Term*$_1$ *Sortqual*-**set** *Dict*

Objective Transform an identifier of a specific sort into AS_1

Parameters

id	The AS_0 identifier to be transformed
totset	A sort *Qual* set containing only the specific sort

Result See *transform-expr* and the introduction to this section

Algorithm

Line 1	Decompose the identifier.
Line 2	Construct the *Qual* of the identifier supposing that it a value identifier.
Line 3	It denotes an explicit quantified value identifier if it is not qualified and it has a *ValueidD* descriptor in *Dict* (line 3) and the specific sort equals the sort for the value identifier (line 5) and the flag (*e*), indicating that the identifier is explicit quantified, is true.
Line 6-8	If the value identifier is not introduced by literal quantification (*val*=**nil**) then return the composite term containing the AS_1 value identifier else
Line 10	Transform the identifier of the associated literal.
Line 12	If the identifier does not represent an explicitly quantified value identifier then all the possible literal *Quals* are extracted.
Line 13-14	If the identifier does not represent a literal then it represents an implicitly quantified value identifier and it must therefore not be qualified.

Line 17-24	Extract or construct the descriptor of the value identifier and denote it by *vald*.
Line 18	If the descriptor already is found in *Dict* (due to use elsewhere) then it is decomposed and
Line 19	There must be at least one legal and common sort for this usage and for the usage elsewhere and the set of legal sorts in the descriptor is then restricted by *totset* (line 22).
Line 24	Else a new descriptor is constructed.
Line 26	Return the AS_1 composite term containing the value identifier, the (single) legal sort and the updated *Dict*.
Line 28-29	If there exist a literal *Qual* of the sort then transform it into AS_1

3.5.2 Character Strings

transform-stringexpr(mk-*Stringterm*$_0$(*q*, *strnm*), *sortset*)(*dict*) $\triangleq$ (3.5.2.1)

```
 1  (let lqualset = all-visible-literals(mk-Stringterm0(q, strnm), sortset)(dict) in
 2   let litsortset = {get-sur(lit) | lit ∈ lqualset} in
 3   if lqualset = {} then
 4     exit("§2.2.2: Character string is not of appropriate sort")
 5    else
 6     if card lqualset = 1 then
 7       (let lqual ∈ lqualset in
 8        let as1lit = make-as1-identifier(lqual)(dict) in
 9        (mk-Ground-term1(as1lit), sortset, dict))
10      else
11       (nil, litsortset, dict))
```

type : *Stringterm*$_0$ *Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ [*Ground-term*$_1$] *Sortqual*-**set** *Dict*

Objective Transform a character string into AS_1

Parameters See *transform-expr* and the introduction to this section

Result See *transform-expr* and the introduction to this section

Algorithm

Line 1	Extract the *Quals* of all the literals which are of a sort in *sortset* and which may be denoted by the *Stringterm*$_0$
Line 2	Extract the *Quals* of their sort
Line 3-4	There must exist at least one possible literal which matches the sort
Line 6-9	If there exist exactly on possible literal (and resulting sort) then extract the *Qual* of the literal (line 7), construct the AS_1 literal (line 8) and return a ground term containing that literal, the sort set containing its sort and the unchanged *Dict*.
Line 11	If there (still) exist more than one possible literal (and sort) then return no AS_1 term and return the set of possible sorts and the unchanged *Dict*

3.5.3 Operators

transform-operatorexpr(mk-*Operatorapp*$_0$(*expr*, *exprlist*), *context*, *tpset*)(*dict*) $\triangleq$ (3.5.3.1)

```
1  (let (qual, nm) = cases expr:
2                      (mk-Id0(q, n)       → (q, n),
3                       mk-Qualop0(q, n) → (q, n),
```

```
4                    T                    → (⟨⟩,nil)) in
5   let level = dict(SCOPEUNIT) in
6   let opqualset = if nm = nil then {} else all-visible-operators(qual, nm, level)(dict) in
7   let legalqualset = extract-legal-operators(exprlist, context, opqualset, tpset)(dict) in
8   let op = build-extract-operator(expr, exprlist) in
9   let (as1 tree', tpset', dict') =
10      if op = nil then
11        (nil, {}, [])
12      else
13        (trap exit with (nil, {}, []) in
14         transform-operatorexpr(op, context, tpset)(dict)) in
15  let fop = build-field-operator(expr, exprlist) in
16  let (as1 tree'', tpset'', dict'') =
17      if fop = nil then
18        (nil, {}, [])
19      else
20        (trap exit with (nil, {}, []) in
21         transform-selectexpr(fop, context, tpset)(dict)) in
22  (card (legalqualset ∪ tpset' ∪ tpset'') = 0
23     → exit("§2.2.2: The operator term cannot be used in this context"),
24   card legalqualset = 0 ∧ card tpset' = 0
25     → (as1 tree'', tpset'', dict''),
26   card legalqualset = 0 ∧ card tpset'' = 0
27     → (as1 tree', tpset', dict'),
28   card (tpset' ∪ tpset'') = 0 ∧ card legalqualset = 1
29     → (let qu ∈ legalqualset in
30        transform-qual-operator(qu, exprlist, context, tpset)(dict)),
31   T → (nil, {get-parent(s-Result(dict(q)))(dict) |
32              q ∈ legalqualset} ∪ tpset' ∪ tpset'', dict)))
```

type : *Operatorapp*$_0$ *Context Sortqual*-**set** → *Dict* → *Expression*$_1$ *Sortqual*-**set** *Dict*

Objective Transform an AS_1 expression which is an operator application or an extract shorthand or a field select shorthand into AS_1

Parameters See *transform-expr* and the introduction to this section

Result See *transform-expr* and the introduction to this section

Algorithm

Line 1-4 Extract the qualifier and the name of the operator. If the operator is a general expression (line 4) then there is no name since it is a shorthand for the extract operator in that case.

Line 5 Let *level* denote the enclosing scopeunit.

Line 6 Extract the *Quals* of all the operators which are possible according to the qualifier and name.

Line 7 Restrict the *Qual* set to contain only those operators which can be used when taking into account the argument list.

Line 8-14 Try the extract operator shorthand. *op* contains the possible result sorts for the extract application.

Line 15-21 Try the field selection shorthand. *fop* contains the possible result sorts for the field selection.

Line 22 If neither the normal operator application nor the extract application nor the field selection can be used in this context then it is an error.

Line 24 If only the field selection can be used in this context then select this shorthand as the result.

Line 26 If only extract application can be used in this context then select this shorthand as the result.

Line 28 If only the normal operator application can be used in this context then extract the operator *Qual* and transform the operator application given that *Qual.*

Line 31 Else the operator is (still) ambiguous and return therefore no AS_1 expression, the sort set consisting of the result sorts of all possible operators in this context joined with the possible sorts for the extract shorthand (*tpset'*) and joined with the sorts for the field select shorthand (*tpset''*).

all-visible-operators(*qual*, *nm*, *level*)(*dict*) $\triangleq$ **(3.5.3.2)**

```
1  {q ∈ dom dict | (∃q')(get-sur(get-sur(q)) ⌢ q' = level) ∧
2                  (∃q'')(get-parent(q'' ⌢ qual)(dict) ⌢ ⟨(OPERATOR, (nm, , ))⟩ = q)}
```

type : *Qual* (*Name*$_0$ | *Quotedop*$_0$) *Qual* → *Dict* → *Operatorqual*-**set**

Objective Extract the *Qual*s of all those operators which are visible in the scopeunit *level* and which includes the qualifier *qual* and which have the name *nm*

Algorithm The resulting set contains all those *Operatorqual*s which are visible (i.e. the scopeunit containing the partial type definition which defines the operator must be part of *level*) and which includes the qualifier in the *Operatorqual.*

transform-operatorterm(**mk**-*Operatorterm*$_0$(*id*, *exprlist*), *context*, *tpset*)(*dict*) $\triangleq$ **(3.5.3.3)**

```
1  (let (qual, nm) = cases id:
2                      (mk-Id0(q, n)       → (q, n),
3                       mk-Qualop0(q, n) → (q, n)) in
4   let level = get-sur(dict(SCOPEUNIT)) in
5   let opqualset = all-visible-operators(qual, nm, get-sur(level))(dict) in
6   let legalqualset = extract-legal-operators(exprlist, context, opqualset, tpset)(dict) in
7   if card legalqualset = 0 then
8     exit("§2.2.2: The operator term cannot be used in this context")
9    else
10    if card legalqualset = 1 then
11      (let qu ∈ legalqualset in
12       transform-qual-operator(qu, exprlist, context, tpset)(dict))
13     else
14      (nil, {get-parent(s-Result(dict(q)))(dict) | q ∈ legalqualset}, dict))
```

type : *Operatorterm*$_0$ *Context Sortqual*-**set** → *Dict* → *Term*$_1$ *Sortqual*-**set** *Dict*

Objective Transform an operator term into AS_1

Parameters See *transform-expr* and the introduction to this section

Result See *transform-expr* and the introduction to this section

Algorithm

Line 1-3 Extract the qualifier and the name of the operator.

Line 4 The level where the operator must be visible is the scopeunit surrounding the sort definition where the operator is used

Line 5 Extract the *Qual*s of all those operators which are visible in the surrounding scopeunit and which may have *qual* as the AS_0 qualifier and which have the name *nm.*

Line 6 Restrict the *Qual* set to contain only those operators which can be used when taking into account the argument list

Line 7	There must exist at least one possible operator term which matches the context.
Line 10-12	If there exist exactly one operator term matching the context then transform it into AS_1 given the right *Qual* (*qu*).
Line 14	If the operator term is (still) ambiguous then return no AS_1 term, but return the result sorts of the possible operators

build-extract-operator(*expr*, *exprlist*) $\triangleq$ (3.5.3.4)

```
1  if is-Qualop0(expr) then
2    nil
3   else
4    (let nm = mk-Name0("EXTRACT", EXCLAMATION) in
5     mk-Operatorapp0(mk-Id0(⟨⟩, nm), ⟨⟨expr⟩ ⌢ exprlist⟩))
```

type: $Expr_0\ Exprlist_0 \rightarrow [Operatorapp_0]$

Objective Construct an extract operator application from its shorthand notation.

example

`a(b,c)`

(a is the formal parameter *expr* and the list `b,c` is the formal parameter *exprlist*) is transformed into

`EXTRACT!(a,b,c)`

If the operator identifier is an infix operator then **nil** is returned since the notation always denotes an operator application in that case

build-field-operator(*var*, *exprlist*) $\triangleq$ (3.5.3.5)

```
1  (let expr = exprlist[len exprlist] in
2   let rest = ⟨exprlist[i] | 1 ≤ i ≤ len exprlist − 1⟩ in
3   if ¬is-Id0(expr) ∧ s-Qualifier0(expr) = ⟨⟩ then
4     nil
5    else
6     (let exprvar = if len rest = 0 then var else build-field-operator(var, rest) in
7      if exprvar = nil then
8        nil
9       else
10       (let mk-Id0(, nm) = expr in
11        mk-Selectexpr0(exprvar, nm))))
```

type: $Expr_0\ Exprlist_0 \rightarrow [Selectexpr_0]$

Objective Construct a field extract shorthand from the shorthand notation with parenthesis

example

`a(b,c)`

(a is the formal parameter *var* and the list b,c is the formal parameter *exprlist*) is transformed into

`a!b!c`

The function is recursive. That is, first `a!b` is constructed then `a!b!c` is constructed

extract-legal-operators(*exprlist*, *context*, *opqualset*, *sortset*)(*dict*) ≙ (3.5.3.6)

```
1  (if opqualset = {} then
2     {}
3   else
4    (let q ∈ opqualset in
5     let (,ts,) =
6         (trap exit with (nil, {}, []) in
7          transform-qual-operator(q, exprlist, context, sortset)(dict)) in
8     let qs = if ts = {} then {} else {q} in
9     qs ∪ extract-legal-operators(exprlist, context, opqualset \ {q}, sortset)(dict)))
```

type : *Exprlist*$_0$ *Context Operatorqual*-**set** *Sortqual*-**set** → *Dict* → *Operatorqual*-**set**

Objective Extract all the operator *Quals* which matches the sort of the actual parameters and the sort of the result and which have a specific name

Parameters

exprlist	The actual parameters
context	The context in which the operator is used (see *transform-expr*)
opqualset	The possible *Quals* which all have the specific name
sortset	The possible result sorts

Result The operator *Quals* which can be used. This set is a subset of *opqualset*

Algorithm

Line 1	When through, return the empty set (the function is recursive).
Line 4	Take an operator *Qual* and test it by
Line 5-7	Trying to transform the operator application, given that operator *Qual*. If this transformation goes well (if *transform-qual-operator* is not trapped) then the set of result sorts (*ts*) is non-empty.
Line 8	If the operator *Qual* can be used then *qs* denotes the *Qual*.
Line 9	Join *qs* with the allowed operator *Quals* from the rest of *opqualset*

transform-qual-operator(*qual*, *exprlist*, *context*, *sortset*)(*dict*) $\triangleq$ (3.5.3.7)

```
 1  (let (, (,parm, tqual)) = qual[len qual] in
 2   if tqual ∉ sortset then
 3     exit("§2.2.2: Result sort of operator is illegal for the context")
 4    else
 5     if len parm ≠ len exprlist then
 6       exit("§5.2: Length of parameter list is not correct for an operator")
 7      else
 8       (let (as1parmlist, d) = transform-actparms(parm, exprlist, context)(dict) in
 9        let as1id = make-as1-identifier(qual)(dict) in
10        let const = (∀p ∈ elems as1parmlist)(is-Ground-term1(p)) in
11        let as1tree =
12            if context ∈ {AXIOMS, MAPPING} then
13              if const then
14                mk-Ground-term1((as1id, as1parmlist))
15               else
16                mk-Composite-term1((as1id, as1parmlist))
17             else
18              if const then
19                mk-Ground-term1((as1id, as1parmlist))
20               else
21                mk-Operator-application1(as1id, as1parmlist) in
22        (as1tree, {tqual}, d)))
```

type: *Operatorqual* (*Exprlist*$_0$ | *Term*$_0$*) *Context Sortqual*-set $\rightarrow$ *Dict* $\rightarrow$
(*Term*$_1$ | *Expression*$_1$) *Sortqual*-set *Dict*

Objective Transform a operator of a specific definition into AS_1

Parameters

Qual	The *Qual* which uniquely identifies the operator
exprlist	The argument list which is a list of terms if the operator occurs in equations. Otherwise it is a list of expressions.
context,sortset	See *transform-expr*

Result See *transform-expr*. The resulting sort set contains only one sort

Algorithm

Line 1	Extract the *Operatorqualelem* from the operator *Qual* and identify the resulting sort by *tqual* and the argument sorts by *parm*
Line 2-6	The result sort must be one of the allowed sorts of this context and the length of the formal parameter list must be equal to the length of the actual parameter list.
Line 8	Transform the actual parameters which are either terms or expressions.
Line 9	Construct the AS_1 identifier of the operator
Line 10	*const* is true if all the actual parameters are ground terms (constants).
Line 11-18	If the operator occurs in the axioms then form an AS_1 term (line 13) else form an AS_1 expression.
Line 22	Return the formed AS_1 term/expression, the result sort of the operator and the *Dict*, possibly updated during the evaluation of the actual parameters (see *transform-expr*)

3.5.3.1 The Spelling Operator

transform-spelling(mk-*Spellingterm*$_0$(mk-*Id*$_0$(*q*, *nm*)), *context*, *sortset*)(*dict*) $\triangleq$ **(3.5.3.1.1)**

```
1  if context ≠ MAPPING ∨ q ≠ ⟨⟩ then
2    exit("§5.4.1.15: Illegal use of the SPELLING operator")
3   else
4    (let stringsort = get-predef-sort("CHARSTRING")(dict) in
5     let qualset = {q ∈ dom dict | is-ValueidD(dict(q)) ∧ s-Mapvalue(dict(q)) ≠ nil ∧
6                                   (VALUE, nm) = q[len q]} in
7     if card qualset = 1 ∧ stringsort ∈ sortset then
8       (let qual ∈ qualset in
9        let mk-ValueidD(litq, , ) = dict(qual) in
10       let (, mk-Name0(str, )) = litq[len litq] in
11       transform-stringexpr(mk-Stringterm0(⟨⟩, str), {stringsort})(dict))
12      else
13       exit("§5.4.1.15: Illegal use of the SPELLING operator"))
```

type : *Spellingterm*$_0$ *Context Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ *Term*$_1$ *Sortqual*-**set** *Dict*

Objective Transform the SPELLING operator into the AS_1 character string literal which it denotes (in the function *transform-mappingaxioms* an axiom is generated for each possible literal of the quantified value name)

Parameters See *transform-expr* and the introduction to this section

Result See *transform-expr* and the introduction to this section

Algorithm

Line 1 The spelling operator must only be used in literal quantification and the contained value identifier must not be quantified

Line 4 Extract the *Qual* of the character string sort

Line 5-7 There must be only one *Qual* (i.e. a unique descriptor), of the contained value name (*nm*), and the predefined character string sort must be allowed in the context.

Line 8-9 Extract the *Qual* of the value name and decompose its descriptor. *litq* denotes the *Qual* of the literal represented by the value name.

Line 10 Extract the spelling of the literal.

Line 11 Construct an AS_0 string term from the literal spelling and transform the string term

3.5.3.2 Export, Import, View, Active

transform-build-in-expression(*expr*, *context*, *sortset*)(*dict*) $\triangleq$ (3.5.3.2.1)

```
 1  if context ≠ EXPRESSION ∧ ¬is-Selectexpr0(expr) ∧ ¬is-Tupleexpr0(expr) then
 2    exit("§5.5.4: Imperative operator cannot be used in constant expression")
 3   else
 4    cases expr:
 5     (mk-Importexpr0(v, exp)
 6        → transform-importexpr(v, exp, sortset)(dict),
 7      mk-Viewexpr0(v, exp)
 8        → transform-viewexpr(v, exp, sortset)(dict),
 9      mk-Nowexpr0()
10        → transform-nowexpr(sortset)(dict),
11      mk-Activeexpr0(,)
12        → transform-activeexpr(expr, sortset)(dict),
13      mk-Selectexpr0(,)
14        → transform-selectexpr(expr, context, sortset)(dict),
15      mk-Tupleexpr0(,)
16        → transform-tupleexpr(expr, context, sortset)(dict),
17      T → transform-pid-build-in(expr, sortset)(dict))
```

type: *Expr*$_0$ *Context Sortqual*-**set** → *Dict* → [*Expression*$_1$] *Sortqual*-**set** *Dict*

Objective Transform one of the special expressions into AS_1

Parameters See *transform-expr* and the introduction to this section

Result See *transform-expr* and the introduction to this section

Algorithm

Line 1-2	Only the cases *Selectexpr*$_0$ or *Tupleexpr*$_0$ of the above alternatives are allowed in constant expressions.
Line 5	Transform an IMPORT expression.
Line 7	Transform a VIEW expression.
Line 9	Transform a NOW expression.
Line 11	Transform a timer ACTIVE expression.
Line 13	Transform a field select expression.
Line 15	Transform a tuple expression.
Line 17	Otherwise, it is a predefined PID expression

transform-importexpr(mk-Id_0(*qu*, *nm*), *expr*, *sortset*)(*dict*) $\triangleq$ (3.5.3.2.2)

```
1  (let level = process-or-service-level(dict(SCOPEUNIT)) in
2   let qualset = {q ∈ dom dict | (∃q', sort)(q' ⌢ qu ⌢ ⟨(IMPORT, (nm, sort))⟩ = q ∧
3                                              (get-sur(q) = level ∨ get-sur(q) = process-level(level)) ∧
4                                              sort ∈ sortset)} in
5   if qualset = {} then
6     exit("§2.2.2: No import variable matches the context")
7    else
8     if card qualset = 1 then
9       (let qual ∈ qualset in
10       let sort = s-Sortqual(dict(qual)) in
11       let importlist = dict(IMPORTLIST) in
12       let imp = dict(IMPLIED) in
13       let nm' = create-unique-name() in
14       let newqual = level ⌢ ⟨(VALUE, nm')⟩,
15           imp' = imp ∪ {(sort, nm')},
16           as1 id = make-as1-identifier(newqual)(dict) in
17       let importlist' = importlist ⌢ ⟨(nm', qual, expr)⟩ in
18       (as1 id, {get-parent(sort)(dict)}, dict + [IMPORTLIST ↦ importlist',
19                                                  IMPLIED    ↦ imp'] ))
20      else
21       (let sortset' = {sort | (∃q ∈ qualset)(dict(q) = mk-ImportD(sort))} in
22        (nil, sortset', dict)))
```

type : Id_0 [$Expr_0$] *Sortqual*-**set** → *Dict* → [$Expression_1$] *Sortqual*-**set** *Dict*

Objective Transform an import expression into its AS_1 representation

Parameters

Id	The identifier of the imported variable
expr	The optional PID expression denoting the exporting process
sortset	See *transform-expr*

Result See *transform-expr* and the introduction to this section. The returned expression is the implicit variable identifier implied by the import expression

Algorithm

Line 1	Extract the level at which the import variable is defined, i.e. a process or a service.
Line 2-4	Extract the import *Quals* of all the import variables which can be specified by means of a qualifier (*qu*), a name (*nm*) (line 2) and a sort *sort* from *sortset* and which are defined in the enclosing service or in the enclosing process (line 3).
Line 5	There must exist at least one possible import variable.
Line 9	If there exist a unique import *Qual* then
Line 9	Denote the *Qual* of that import variable by *qual.*
Line 10	Denote the *Qual* of its sort by *sort.*
Line 11	Denote the current list of import expressions used elsewhere in the surrounding expression by *importlist.*
Line 12	Denote the current set of implicit variables used in the process body by *imp.*
Line 13-14	Create a unique AS_0 variable name and construct its *Qual.*
Line 15	Update the current set of implicit variables with the pair of its sort (*sort*) and its new name.
Line 16	Construct the AS_1 identifier of the implicit variable.

Line 17	Update the current list of import expressions.
Line 18	Return the AS_1 identifier and the *Dict* containing the updated importlist and implicit variable set.
Line 21	If the import expression is (still) ambiguous, then return the set of possible sorts

transform-viewexpr(*id*, *expr*, *sortset*)(*dict*) $\triangleq$ (3.5.3.2.3)

```
1  (let mk-Id0(qu, nm) = id in
2   let pqual = process-or-service-level(dict(SCOPEUNIT)) in
3   let qset = {q ∈ dom dict | (let qual = pqual ⌢ ⟨(VIEW, q)⟩ in
4                               qual ∈ dom dict ∧
5                               get-parent(s-Qual(dict(qual)))(dict) ∈ sortset ∧
6                               (∃q')(q' ⌢ qu ⌢ ⟨(VALUE, nm)⟩ = q))} in
7   if card qset ≠ 1 then
8     exit("§2.2.2: One and only one revealed variable must match the view expression")
9     else
10    (let q ∈ qset in
11     let inst = get-predef-sort("PID")(dict) in
12     let as1id = make-as1-identifier(q)(dict),
13         (as1expr, , d) = transform-expr(expr, EXPRESSION, {inst})(dict) in
14     let as1tree = mk-View-expression1(as1id, as1expr) in
15     (as1tree, {s-Sortqual(dict(q))}, d)))
```

type: *Varid*$_0$ *Expr*$_0$ *Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ *View-expression*$_1$ *Sortqual*-**set** *Dict*

Objective Transform a View expression into AS_1

Parameters

id	The view variable
expr	The Pid expression denoting the revealing process
sortset	See *transform-expr*

Result See *transform-expr*

Algorithm

Line 1	Decompose the view variable identifier.
Line 2	Extract the level on which the view variable is defined.
Line 3-6	Let *qset* denote set which matches the view expression. The set consist of those variable *Quals* which are viewed in the process (line 3-4), are of an appropriate sort (line 5) and which includes the qualifier and name specified in the view expression (line 6).
Line 11	Extract the *Qual* of the PID sort.
Line 12	Construct the AS_1 identifier of the revealing variable (which is the same as the identifier of the viewed variable).
Line 13	Transform the PID expression.
Line 14	Construct the AS_1 view expression.
Line 15	Return the AS_1 view expression, the sort of the variable and the (possibly) updated *Dict*

transform-nowexpr(*sortset*)(*dict*) $\triangleq$ (3.5.3.2.4)

```
1  (let tqual = get-predef-sort("TIME")(dict) in
2   if tqual ∈ sortset then
3     (mk-Now-expression1(), {tqual}, dict)
4     else
5     exit("§5.5.4.1: NOW expression must be used where TIME values are allowed"))
```

type: *Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ *Now-expression*$_1$ *Sortqual*-**set** *Dict*

Objective Transform a NOW expression into AS_1

Parameters See *transform-expr*

Result See *transform-expr*

Algorithm

Line 1	Extract the *Qual* denoting the TIME sort.
Line 2	If this sort is allowed at this place
Line 3	Then return the AS_1 now expression, the TIME sort, and the unchanged *Dict*

transform-activeexpr(mk-*Activeexpr*$_0$(*id*, *actparm*), *sortset*)(*dict*) $\triangleq$ (3.5.3.2.5)

```
1  (let qual = get-visible-qual(id, SIGNAL)(dict),
2       bqual = get-predef-sort("BOOLEAN")(dict) in
3   let as1id = make-as1-identifier(qual)(dict) in
4   if ¬is-TimerD(dict(qual)) then
5     exit("§2.8: Identifier in timer active expression does not denote a timer")
6    else
7     (let mk-TimerD(tplist, ) = dict(qual) in
8      if bqual ∈ sortset ∧ len actparm = len tplist then
9        (let (as1actparm, d) = transform-actparms(tplist, actparm, EXPRESSION)(dict) in
10        let as1tree = mk-Timer-active-expression1(as1id, as1actparm) in
11        (as1tree, {bqual}, d))
12       else
13        exit("§2.8: Illegal timer active expression")))
```

type : *Activeexpr*$_0$ *Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ *Timer-active-expression*$_1$ *Sortqual*-**set** *Dict*

Objective Transform a timer active expression into AS_1

Parameters

id	The timer identifier
actparm	The actual parameters
sortset	See *transform-expr*

Result See *transform-expr*

Algorithm

Line 1-2	Extract the *Qual* of the timer and the *Qual* of the boolean sort.
Line 3	Construct the AS_1 identifier of the timer.
Line 4	The AS_0 identifier must denote a timer (not a signal).
Line 7	Let *tplist* denote the list of sorts of the parameters.
Line 8	The boolean sort must be an allowed result sort in this context and the length of the actual parameter list must be equal to the length of the sort list.
Line 9	Transform the actual parameters into AS_1.
Line 10-11	Construct the AS_1 timer active expression and return it together with the boolean sort and the (possibly) updated *Dict*

transform-selectexpr(mk-*Selectexpr*$_0$(*v*, *nm*), *context*, *sortset*)(*dict*) $\triangleq$ (3.5.3.2.6)

```
1  (let mk-Name0(str, ) = nm in
2   let opnm = mk-Name0(str ⌢ "EXTRACT", EXCLAMATION) in
3   let opid = mk-Id0(⟨⟩, opnm) in
4   transform-expr(mk-Operatorapp0(opid, ⟨v⟩), context, sortset)(dict))
```

type : *Selectexpr*$_0$ *Context Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ *Expression*$_1$ *Sortqual*-**set** *Dict*

Objective Transform a field select expression into AS_1

Parameters See *transform-expr*

Result See *transform-expr*

Algorithm

Line 1 Let *str* denote the spelling of the field.

Line 2 Form the name of the EXTRACT! operator, which operates on the specified field.

Line 3-4 Form the identifier of the operator, construct an operator application given the left-hand expression as parameter and transform it the usual way

transform-tupleexpr(**mk-***Tupleexpr*$_0$(*q*, *exprlist*), *context*, *sortset*)(*dict*) $\triangleq$ **(3.5.3.2.7)**

```
1  (let opid = mk-Id0(q, mk-Name0("MAKE", EXCLAMATION)) in
2   transform-expr(mk-Operatorapp0(opid, exprlist), context, sortset)(dict))
```

type: *Tupleexpr*$_0$ *Context Sortqual*-**set** → *Dict* → *Expression*$_1$ *Sortqual*-**set** *Dict*

Objective Transform a tuple expression (a structure primary) into AS_1

Parameters See *transform-expr*

Result See *transform-expr*

Algorithm

Line 1 Form the identifier of the MAKE! operator.

Line 2 Construct an operator application given the expression list as parameter list and transform it the usual way

transform-pid-build-in(*expr*, *sortset*)(*dict*) $\triangleq$ **(3.5.3.2.8)**

```
 1  (let pqual = get-predef-sort("PID")(dict) in
 2   if pqual ∈ sortset then
 3     ( cases expr:
 4       (mk-Parentexpr0()
 5          → (mk-Parent-expression1(), {pqual}, dict),
 6        mk-Offspringexpr0()
 7          → (mk-Offspring-expression1(), {pqual}, dict),
 8        mk-Senderexpr0()
 9          → (mk-Sender-expression1(), {pqual}, dict),
10        mk-Selfexpr0()
11          → (mk-Self-expression1(), {pqual}, dict)))
12    else
13     exit("§5.5.4.3: PID expression is used in wrong context"))
```

type: *Expr*$_0$ *Sortqual*-**set** → *Dict* → *Pid-expression*$_1$ *Sortqual*-**set** *Dict*

Objective Transform one of the predefined PID expressions into AS_1

Parameters See *transform-expr*

Result See *transform-expr*

Algorithm

Line 1 Extract the *Qual* of the PID sort.

Line 2 The PID sort must be one of the allowed sorts at this place.

Line 4-10 Return the *Qual* of the PID sort (*pqual*), the unchanged *Dict* and

Line 4 A parent expression or

Line 6 An offspring expression or

Line 8	A sender expression or
Line 10	A self expression

3.5.4 Conditional Expressions

transform-condexpr(mk-*Condexpr*$_0$(*bexpr*, *expr*1, *expr*2), *context*, *sortset*)(*dict*) $\triangleq$ (3.5.4.1)

```
1  (let boolqual = get-predef-sort("BOOLEAN")(dict) in
2   let (as1, , d) = transform-expr(bexpr, context, {boolqual})(dict) in
3   let (,s', d') = transform-expr(expr1, context, sortset)(d) in
4   let (,s'',) = transform-expr(expr2, context, sortset)(d') in
5   if s' ∩ s'' = {} then
6     exit("§2.2.2: Conditional expression does not match the context")
7    else
8     if card (s' ∩ s'') = 1 then
9       (let sort ∈ s' ∩ s'' in
10       let (as1 unique, , dict') = transform-expr(expr1, context, {sort})(dict) in
11       let (as1 unique', , dict'') = transform-expr(expr2, context, {sort})(dict') in
12       let as1 tree =
13           if context ∈ {AXIOMS, MAPPING} then
14             mk-Conditional-term1(as1, as1 unique, as1 unique')
15            else
16             mk-Conditional-expression1(as1, as1 unique, as1 unique') in
17       let as1 tree' =
18           if is-Ground-term1(as1) ∧ is-Ground-term1(as1 unique) ∧ is-Ground-term1(as1 unique')
19             then mk-Ground-term1(as1 tree)
20             else as1 tree in
21       (as1 tree', {sort}, dict''))
22      else
23       (nil, s' ∩ s'', dict))
```

type : *Condexpr*$_0$ *Context Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ [*Term*$_1$ | *Expression*$_1$] *Sortqual*-**set** *Dict*

Objective Transform a conditional expression or a conditional term into AS_1

Parameters See *transform-expr*

Result See *transform-expr*

Algorithm

Line 1	Extract the *Qual* of the boolean sort.
Line 2	Transform the boolean condition.
Line 3	Transform the "then" expression/term.
Line 4	Transform the "else" expression/term.
Line 5-6	There must exist at least one resulting sort which can be used.
Line 8	If there is exactly one sort which can be used then
Line 9	Let *sort* denote the sort.
Line 10-11	Transform the expressions again, this time with the unique sort given as argument.
Line 12-16	Form the AS_1 expression or term.
Line 17-20	If both the condition, the "then" part and the "else" part are ground terms then the resulting AS_1 expression is a ground term.
Line 20	Return the AS_1 expression/term, the sort of the conditional expression/term and the (possibly) updated *Dict*

3.5.5 Infix and Prefix operators

transform-monadexpr(mk-*Monadexpr*$_0$(*op*, *expr*), *context*, *sortset*)(*dict*) $\triangleq$ (3.5.5.1)

```
1  (let qinfix = mk-Qualop0(⟨⟩, mk-Quotedop0(op)) in
2   let operator =
3       if context ∈ {MAPPING, AXIOMS}
4         then mk-Operatorterm0(qinfix, ⟨expr⟩)
5         else mk-Operatorapp0(qinfix, ⟨expr⟩) in
6   transform-expr(operator, context, sortset)(dict))
```

type: *Monadexpr*$_0$ *Context Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ [*Expression*$_1$ | *Term*$_1$] *Sortqual*-**set** *Dict*

Objective Transform a prefix operator application into AS$_1$

Parameters See *transform-expr*

Result See *transform-expr*

Algorithm

Line 1 From the operator symbol, construct the qualified prefix operator (the qualifier is empty).

Line 2-5 Construct the operator term or operator application depending of whether it is used in the axioms or in the expressions.

Line 6 Transform the constructed term/expression in the usual way

transform-infixexpr(mk-*Infixexpr*$_0$(*expr*1, *op*, *expr*2), *context*, *sortset*)(*dict*) $\triangleq$ (3.5.5.2)

```
1  (let qinfix = mk-Qualop0(⟨⟩, mk-Quotedop0(op)) in
2   let operator =
3       if context ∈ {MAPPING, AXIOMS}
4         then mk-Operatorterm0(qinfix, ⟨expr1, expr2⟩)
5         else mk-Operatorapp0(qinfix, ⟨expr1, expr2⟩) in
6   transform-expr(operator, context, sortset)(dict))
```

type: *Infixexpr*$_0$ *Context Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ [*Expression*$_1$ | *Term*$_1$] *Sortqual*-**set** *Dict*

Objective Transform an infix operator application into AS$_1$

Parameters See *transform-expr*

Result See *transform-expr*

Algorithm

Line 1 From the operator symbol, construct the qualified infix operator (the qualifier is empty).

Line 2-5 Construct the operator term or operator application depending of whether it is used in the axioms or in the expressions.

Line 6 Transform the constructed term/expression in the usual way

3.6 Visibility Handling

all-visible-sorts(*dict*) $\triangleq$ (3.6.1)

```
1  {qual ∈ dom dict | is-SortD(dict(qual)) ∧
2                     (∃q ∈ Qual)(get-sur(qual) ⌢ q = dict(SCOPEUNIT))}
```

type: *Dict* $\rightarrow$ *Sortqual*-**set**

Objective Extract the set of sort *Quals*, which are visible at a given place indicated by SCOPEUNIT, from *Dict*

Algorithm Return all the *Quals* which denote a sort (line 1) and for which there exist a partial *Qual* (q) such that the *Qual* of the scopeunit which defines the sort, when joined with q, can form the *Qual* of the current scopeunit

all-visible-literals(*id*, *sortset*)(*dict*) $\triangleq$ (3.6.2)

```
1  (let (qual, nm) = cases id:
2                      (mk-Id0(q, n)          → (q, n),
3                       mk-Stringterm0(q, n) → (q, n)) in
4   {q ∈ dom dict | (∃squal ∈ sortset)(squal ⌢ ⟨(LITERAL, nm)⟩ = q) ∧
5                   (∃qu)(get-parent(qu ⌢ qual)(dict) ⌢ ⟨(LITERAL, nm)⟩ = q)})
```

type: (Id_0 | $Stringterm_0$) *Sortqual*-**set** → *Dict* → *Qual*-**set**

Objective Extract the set of all literals *Quals* which are visible at a given place and which can be denoted by a certain AS_0 identifier and which are of some specific sorts

Parameters

id	The AS_0 identifier which can denote the literal
sortset	The resulting literal *Quals* are all of a sort contained in *sortset*

Algorithm

Line 1-3	Extract the qualifier and the name of the identifier (or string).
Line 4	Return all the *Quals* which denotes literals defined in one of the possible sorts and which includes the qualifier.

all-input-signals(*qual*)(*dict*) $\triangleq$ (3.6.3)

```
1  (let qual' = process-or-service-level(qual) in
2    cases dict(qual'):
3     (mk-ServiceD(, , , inp, ) → inp,
4      mk-ProcessD(, inp, , ) → inp))
```

type: *Qual* → *Dict* → *Signalqual*-**set**

Objective Extract the *Quals* of the input signals (including timers)

Parameters

qual	The *Qual* denoting the level (scopeunit) where the input signals are required

Algorithm

Line 1	In the case of a procedure, modify the level by selecting the surrounding process or service.
Line 3	If the scopeunit is a service then return all the input signals.
Line 4	If the scopeunit is a process then return all the input signals

import-export-signals(*dict*) $\triangleq$ (3.6.4)

```
1  (let (emptyq, , ) = dict(GLOBALNAMES) in
2   let (exportinput, exportoutput) = extract-exports(dict) in
3   let importset = {(tq, nm) ∈ dom exportmap | (∃q ∈ dom dict)(process-level(q) = dict(SCOPEUNIT) ∧
4                                                              (IMPORT, (nm, tq)) = q[len q])} in
5   let importinput =
6       {xrq | (∃(tq, nm) ∈ importset)
7               ((let mk-SignalnamesD(, , xr') = exportmap((tq, nm)) in
8                 get-sur(tq) ⌢ ⟨(SIGNAL, xr')⟩ = xrq))} in
9   let importoutput =
10      {xqq | (∃(tq, nm) ∈ importset)
11              ((let mk-SignalnamesD(, xq, ) = exportmap((tq, nm)) in
12                get-sur(tq) ⌢ ⟨(SIGNAL, xq)⟩ = xqq))} in
13  (exportinput ∪ importinput ∪ {emptyq}, exportoutput ∪ importoutput))
```

type: *Dict* → *Signalqual*-**set** *Signalqual*-**set**

Objective Extract and return the *Quals* of all the implicit signals associated to import and export for a process and return the emptyq signal.

Algorithm

Line 1 Extract the *Qual* of the emptyq signal used in enabling condition/continuous signal.

Line 2 Construct the *Quals* of all the input signals and the *Quals* of all the output signals used when a process exports variables (i.e. the xtQUERY input signals and the xtREPLY output signals).

Line 3-4 Extract the *NameclosureDs* (see the definition of *exportmap*) of all the imported variables in a process. The set consists of all those pairs of sort (*tq*) and name (*nm*) for which there exist an imported variable defined in the process or in a contained service *nm* which are of the sort *tq*.

Line 5-8 Construct the *Quals* of all the xtREPLY signals used by a process. The set consist of all those xtREPLY *Quals* which can be formed from a xtREPLY name contained in a *SignalnamesD* descriptor of a *NameclosureD* in *importset*.

Line 9-12 Do the same for the xtQUERY signals used by a process.

Line 13 Return the union of the *Quals* of the xtQUERY signals, of the xtREPLY signals and of the emptyq signal. The emptyq signal is not returned as an output signal because output signals are collected in order to construct implicit signal routes and no signal route contains the emptyq signal.

extract-exports(*dict*) $\triangleq$ **(3.6.5)**

```
1  (let vqualset = {qual ∈ dom dict | process-level(qual) = dict(SCOPEUNIT) ∧ is-VarD(dict(qual))} in
2   let closset =
3       {(tq, nm) ∈ dom dict | (∃q ∈ vqualset)
4                                  ((let mk-VarD(t, , exp, , ) = dict(q) in
5                                    t = tq ∧ (VALUE, nm) = q[len q] ∧
6                                    exp = EXPORTED))} in
7   let inputsig =
8       {qual | (∃(tq, nm) ∈ closset)
9                ((let mk-SignalnamesD(, xq, ) = exportmap((tq, nm)) in
10                 get-sur(tq) ⌢ ⟨(SIGNAL, xq)⟩ = qual))},
11      outputsig =
12      {qual | (∃(tq, nm) ∈ closset)
13               ((let mk-SignalnamesD(, , xr) = exportmap((tq, nm)) in
14                 get-sur(tq) ⌢ ⟨(SIGNAL, xr)⟩ = qual))} in
15  (inputsig, outputsig))
```

type : *Dict* → *Signalqual*-**set** *Signalqual*-**set**

Objective Extract and return the *Quals* of all the xtQUERY and xtREPLY signals used by an exporting process

Algorithm

Line 1	Extract the *Quals* of the process variables.
Line 2-6	Construct, from the variable set, the *NameclosureD*s used by the process, i.e. the pairs of sort *tq* and variable name *nm* from *Exportmap* for which the variable of the sort *tq* is exported.
Line 7-10	From this set of *NameclosureD*s construct the *Quals* of the xtQUERY signals (the same way as done for the xtREPLY signals in the function *import-export-signals*).
Line 11-14	Do the same for the xtREPLY signals.
Line 15	Return the input signals and the output signals

get-visible-qual(*entity*, *entityclass*)(*dict*) $\triangleq$ **(3.6.6)**

```
1  if entity = ENV then
2    ENV
3   else
4    (let (qual, nm) = cases entity:
5                        (mk-Id0(q, n)          → (q, n),
6                         mk-Stringterm0(q, n) → (q, n),
7                         mk-Qualop0(q, n)     → (q, n)) in
8     let level = dict(SCOPEUNIT) in
9     if (∃q ∈ dom dict)((∃q')(q' ⌢ qual ⌢ ⟨(entityclass, nm)⟩ = q) ∧
10                         (∃q'')(get-sur(q) ⌢ q'' = level)) then
11      (let q ∈ dom dict be s.t. (∃q')(q' ⌢ qual ⌢ ⟨(entityclass, nm)⟩ = q) ∧
12           (∃q'')(get-sur(q) ⌢ q'' = level ∧
13                  ¬(∃qu)(get-sur(q) ⌢ qu = level ∧ len qu < len q'')) in
14       if is-ErrorD(dict(q)) then
15         exit("§2.2.2: Synonym or generator or signal list is recursive defined")
16         else
17          q)
18       else
19        exit("§2.2.2: Identifier is not visible"))
```

type : (*Id*0 | *Stringterm*0 | *Qualop*0 | ENV) *Quot* → *Dict* → (*Qual* | ENV)

Objective Construct the *Qual* for an AS0 identifier if the identifier is visible

Parameters

entity	The identifier (or string or infix operator) for which a *Qual* is constructed
entityclass	The entity class of the identifier (see §2.2.2 of Z.100). The entity class VALUE includes synonyms, variables and value identifiers. The function is not used for constructing *Quals* of literals and operators (due to special visibility rules).

Algorithm

Line 1	If *entity* is ENV then return ENV (see in *transform-channeldef*).
Line 4-7	Extract the qualifier and name of the identifier (or string or infix operator).
Line 9-10	If there exist a *Qual* in *Dict* which includes the qualifier and the pair of entity class and name in the right-most part (line 9) and the scopeunit denoted by the $qualifier_0$ is defined is visible then
Line 11-13	Let q denote such a *Qual* (line 11-13) and make the selection unique by expressing that q must denote the "nearest" *Qual* with respect to the current level (line 13)
Line 14-15	The identifier must not be recursively defined.

signal-qual(*sigid*)(*dict*) $\triangleq$ (3.6.7)

```
 1  (let qual = (trap exit with nil in
 2               {get-visible-qual(sigid, SIGNAL)(dict)}) in
 3   if qual ≠ nil then
 4     qual
 5    else
 6     (let mk-Id0(q, nm) = sigid in
 7      let qualelem = q ⌢ ⟨(SIGNAL, nm)⟩ in
 8      let level = dict(SCOPEUNIT) in
 9      let qualset = {qu ∈ dom dict | (∃qua)(qua ⌢ qualelem = qu) ∧
10                                     (∃q', q'')(q' ⌢ q'' = level ∧ signaldeflevel(qu) = q')} in
11      if card qualset ≠ 1 then
12        exit("§2.2.2: Signal identifier denotes more than one (sub) signal")
13       else
14        (let qual' ∈ qualset in
15         qual')))
```

type: $Sigid_0 \rightarrow Dict \rightarrow Signalqual$

Objective Construct and return the *Qual* of an AS_0 signal identifier. The reason why the function *get-visible-qual* cannot be used directly for signals is that the visibility rules for signals are a little bit more complicated due to the subsignal concept

Parameters

sigid	The AS_0 signal identifier

Algorithm

Line 1-3	If *sigid* is not a subsignal then the *Qual* can be found using the function *get-visible-qual.*
Line 7	Construct the partial *Qual* which includes the *Qualelem* containing the signal name
Line 8	*level* denotes the scopeunit where *sigid* is used.
Line 9-10	The complete set of possible *Quals* for the signal identifier is the *Quals* representing the sub-signals. The set of *Quals* for the sub-signals are those which have the partial *Qual* as the ending part (line 9) and which are contained in a visible signal definition q'.

Line 11-15 If there is one and only one possible *Qual* for *sigid* then return the *Qual*

signaldeflevel(*qual*) $\triangleq$ (3.6.8)

```
1  if s-Kind(qual[len qual]) = SIGNAL then
2    signaldeflevel(get-sur(qual))
3   else
4    qual
```

type : *Signalqual* → *Signalqual*

Objective From a *Qual* denoting a scopeunit which is a (sub) signal definition, construct the *Qual* of the scopeunit where the outermost signal definition is defined

Parameters

Algorithm

Line 1-2 If the last *Qualelem* contains the entity class SIGNAL then the *Qual* (still) denotes a signal definition and the *Qualelem* is removed before applying the function again

get-visible-variable(*id*, *sortset*, *export*)(*dict*) $\triangleq$ (3.6.9)

```
1  (let qualset = all-variables-and-synonyms(id, EXPRESSION, sortset)(dict) in
2   let qualset' =
3       {qual ∈ qualset |  cases dict(qual):
4                            (mk-VarD(, e, , , )
5                               → export = false ∨ e = EXPORT,
6                             mk-SynD(, )
7                               → false)} in
8   if card qualset' = 1 then
9     (let qual ∈ qualset' in
10     qual)
11    else
12     exit("§2.2.2: One and only one variable must match the context"))
```

type : Id_0 *Sortqual*-**set** *Bool* → *Dict* → *Qual*

Objective Extract the *Qual* of a variable. *get-visible-variable* is applied only where synonyms are not allowed (e.g. assignment, IN/OUT parameters etc.)

Parameters

id	The variable identifier.
sortset	The set of sorts which are allowed in the context.
export	A flag indicating whether the variable is used in an export action.

Result The *Qual* which is unique for the context.

Algorithm

Line 1	Construct the set of *Qual*s consisting of those variables and synonyms which matches one of the sorts in *sortset*.
Line 2-6	Restrict the set to contain only variables and if *id* is used in an export action then the variables in the set must be exported.
Line 1	The set must contain a unique *Qual*

3.7 Transformation of Procedure- and Process Graph

In this section, the body of a process or a procedure is transformed. The result is a *Process-graph*$_1$ or a *Procedure-graph*$_1$ and a *Dict* which carries information of the implicit variables and output signals used in the body. If a process contains services, then also some AS$_1$ definitions are returned. The entry function transforming a process body is *transform-process-body* and the entry function which transforms a procedure body is *transform-body* (which also is used by *transform-process-body* if the process contains no service decomposition). For reasons of simplicity, transitions are only traversed once which means that the short-hands in transitions (i.e. option nodes, dash nextstate and import/export) are expanded "on the fly" while transitions are transformed.

The transformation of a body consist of the following steps:

1. Insertion of terminators in answers of decisions and options such that every AS$_0$ *Transition*$_0$ contains a *Termstmt*$_0$

2. Construction of a *Labeldict*, i.e. a collection of the labels used in the body and their associated transition

3. Construction of a *Statedict* i.e. the list of states is transformed into a map which is a more appropriate representation when the states are transformed. During this construction, multiple appearence of states and asterisk states are removed.

4. Expansion of asterisk inputs and asterisk save. Also the implicit transitions are added. As the signals implied by export and the empty queue signal are shared by all services, the transitions for those signals is not added until the services have been merged.

5. If a process contains services, the *Statedict* of all services are merged ("state exploded") into a *statedict*, where each entry is a new state name.

6. Expansion of enabling condition and continuous signal. The expansion is achieved by modifying the constructed *Statedict*

7. Transformation of *Statedict* and the contained transitions into a *Process-graph*$_1$ or *Procedure-graph*$_1$. During this transformation, new entries implied by import expressions are added to *Statedict*

transform-process-body(*body*)(*dict*) $\triangleq$ (3.7.1)

```
1  if is-Body0(body) then
2    (let (as1 body, bdict) = transform-body(body)(dict) in
3     ({},as1 body, bdict, []))
4   else
5    (let mk-Decomposition0(decll) = body in
6     let (as1 dclset, sdict) = transform-decllist(decll)(dict) in
7     let connectmap = service-connectmap(decll, {}, [])(dict) in
8     if is-wf-servicesignals(dict) then
9       (let (as1 body, dict') = transform-decomposition-body(dict) in
10       (as1 dclset, as1 body, sdict + dict', connectmap))
11     else
12      exit("§4.10.2: Service signals not well-formed"))
```

type: *Processbody*$_0$ $\rightarrow$ *Dict* $\rightarrow$ *Decl*$_1$-set *Process-graph*$_1$ *Dict ProcessconnectionD*

Objective Transform a *Processbody*$_0$ into its AS$_1$ representation

Parameters

body The process body

Result A list of AS_1 definitions (the list is non-empty if the process contains services), the AS_1 process body, a *Dict* carrying information about implicit variables, output signals etc. used in the process body and the connections of signal routes with service signalroutes for the process.

Algorithm

Line 1-3 If the process does not contain services then transform the state body contained in the process (like for procedures) else

Line 5 Let *decll* denote the service definitions and service signal route definitions.

Line 6 Transform the services and signal routes. During this transformation, the services are replaced by the definitions they contain. Their state body is represented as a *Statedict* in the descriptors of the services (contained in *sdict*).

Line 8 The valid input signal set of the services must be disjoint and no signal in a signal routes external to the process may be a high priority signal.

Line 9 Construct the AS_1 process body from the *Statedicts* contained in the service descriptors of *dict*.

Line 10 Return the AS_1 definitions from the services, the process body and the *Dict* contributions from the definitions and from the implicit variables, output signals etc. used in the services

transform-body(**mk-***Body*$_0$(*trans*, *statelist*))(*dict*) $\triangleq$ (3.7.2)

```
1  (let trans' = insert-trans-term(trans) in
2   let statelist' = ⟨insert-state-term(statelist[i]) | 1 ≤ i ≤ len statelist⟩ in
3   let labeldict = build-trans-labeldict(trans')([]) in
4   let labeldict' = build-state-labeldict(statelist')(labeldict) in
5   let statedict = build-statedict(statelist', dict(SCOPEUNIT))([]) in
6   let statedict' = remove-asterisk-input-and-save(statedict)(dict) in
7   let statedict'' = remove-cont-enable-from-statelist(dom statedict)(dict, statedict') in
8   let dict' = dict + [LABELDICT ↦ labeldict',
9                       STATEDICT ↦ statedict''] in
10  let (trans1, dict'') = transform-transition(nil, trans', ⟨⟩)(dict') in
11  let (statebody1, dict''') = transform-statelist({})(dict'') in
12  let transition1 = add-varinit(trans1, dict''') in
13  if is-ProcedureD(dict(dict(SCOPEUNIT))) then
14    (mk-Procedure-graph1(mk-Procedure-start-node1(transition1), statebody1), dict''')
15   else
16    (mk-Process-graph1(mk-Process-start-node1(transition1), statebody1), dict'''))
```

type: *Body*$_0$ $\rightarrow$ *Dict* $\rightarrow$ (*Procedure-graph*$_1$ | *Process-graph*$_1$) *Dict*

Objective Transform a state body of a process or procedure

Parameters An AS_0 body containing

trans The initial transition

statelist The list of states

Result An AS_1 procedure- or process graph and a *Dict* carrying information about implicit variables and output signals

Algorithm

Line 1 Insert terminators in answers of decisions and options in the initial transition

Line 2 Insert terminators in answers of decisions and options in the state list.

Line 3	Collect information about the labels (connectors) used in the initial transition.
Line 4	Collect information about the labels used in the statelist. *labeldict'* reflects the association between the labels used in the process or procedure body and their following transitions (see the definition of *Labeldict*).
Line 5	Transform the statelist into a map from state names into a *StateD* descriptor (see the definition of *Statedict* in *Quotdict*). Multiple appearence of states and asterisk states are merged during this transformation.
Line 6	Expand asterisk inputs, asterisk saves and implicit transitions in the *Statedict*.
Line 7	Expand any enabling condition and continuous signal in the constructed *Statedict*. After this expansion, the *Statedict* contains *ContenablestateD* descriptors for the states which are removed. The new states are represented by a *StateD* descriptor.
Line 8	Include the *Labeldict* and the *Statedict* in the *Dict* such that the transformation functions only need one dictionary parameter (instead of three).
Line 10	Transform the initial transition into AS_1.
Line 11	Transform the statelist into a set of AS_1 states.
Line 12	Modify the initial transition to include the initiation of the implicit variables used in the process.
Line 13-16	Return an AS_1 procedure graph if the state body belongs to a procedure, otherwise return an AS_1 process graph. In the returned *Dict* (*dict'''*) only the OUTSIGNALS and IMPLIED entries from *Quotdict* are used

add-varinit(*trans*, *dict*) $\triangleq$ **(3.7.3)**

```
1  (let qualset = {qual ∈ dom dict | is-VarD(dict(qual)) ∧
2                                    get-sur(qual) = dict(SCOPEUNIT)} in
3   add-default-assign(trans, qualset)(dict))
```

type: $Transition_1$ *Dict* $\rightarrow$ $Transition_1$

Objective Modify the initial transition to include variable initiations.

Parameters

trans	The AS_1 initial transition.
dict	The *Dict* containing the relevant *VarD* descriptors and containing the AS_0 name of the variable attached to continuous signal (the name is the same for all processes)

Result The modified AS_1 transition.

Algorithm

Line 1-2	Extract the set of all those *Quals* which denote local variables.
Line 3	Modify the transition to include default assignments to the local variables.

add-default-assign(*trans*, *descrset*)(*dict*) $\triangleq$ **(3.7.4)**

```
 1  if descrset = {} then
 2    trans
 3   else
 4    (let qual ∈ descrset in
 5     let trans' = add-default-assign(trans, descrset \ {qual})(dict) in
 6     let mk-VarD(,,, expr1, qual) = dict(qual) in
 7     if expr1 = nil then
 8       trans'
 9      else
10       (let as1id = make-as1-identifier(qual)(dict) in
11        let stmt = mk-Task-node1(mk-Assignment-statement1(as1id, expr1)) in
12        let mk-Transition1(actl, term) = trans' in
13        let trans'' = mk-Transition1(stmt ⌢ actl, term) in
14        if is-SyntypeD(dict(qual)) then
15          (let range = s-Range-condition1(dict(qual)) in
16           let mk-Range-condition1(, cset) = range in
17           if cset = {} then
18             trans''
19            else
20             (let des = mk-Decision-node1(expr1, {mk-Decision-answer1(range, trans'')},
21                                                 mk-Else-answer1(trans')) in
22              mk-Transition1(⟨⟩, des)))
23         else
24          trans''))
```

type : *Transition*$_1$ *Qual*-**set** $\rightarrow$ *Dict* $\rightarrow$ *Transition*$_1$

Objective Modify the initial transition to include the default assignments for the variables local to a scopeunit. As there also are *VarD* descriptors for the implicit variables attached to export variables, the implicit export variables are automatically initialized.

Parameters

trans	The transition which should be modified.
descrset	The *Quals* of the local variables.

Result The modified transition.

Algorithm

Line 1	When through, do not modify the transition any more.
Line 4	Let *qual* denote an element in the set of *Quals*.
Line 5	Construct the transition which is *trans* modified to include variable initiation for the rest of the *Quals*.
Line 6	Decompose the descriptor of the variable in order to reveal the initial expression (*expr*$_1$) and the variable sort. Note that *qual* is not used for deriving the initial expression (this has already been done).
Line 7	If the descriptor do not contains an initial expression then return do not modify the transition.
Line 10	Let *as*$_1$*id* denote the AS$_1$ identifier of the variable.
Line 11	Let *stmt* denote the task which contains the default assignment.
Line 12-13	Let *trans''* denote the modified transition.
Line 14-17	If the variable sort is a syntype then a decision must be constructed.
Line 15	Let *range* denote the AS$_1$ range condition for the syntype.
Line 16-17	If the set of *Condition-items* in *range* is empty then there is no need to construct a syntype and the modified transition is therefore returned.

Line 20-22 Return a new transition which consist of an empty action list and an AS_1 decision able of performing the default assignment only in the case where the range condition is satisfied.

3.7.1 Insertion of Terminators in Answers

insert-state-term(mk-*Statebody*$_0$(*idl*, *stlist*, *tnm*)) $\triangleq$ (3.7.1.1)

```
1  (let stlist' = ⟨insert-spec-term(stlist[i]) | 1 ≤ i ≤ len stlist⟩ in
2   mk-Statebody0(idl, stlist', tnm))
```

type: *Statebody*$_0$ → *Statebody*$_0$

Objective Insert terminators in answers in decisions and options contained in an AS_0 state

Parameters The state containing

idl The list of state names for the state

Result The modified AS_0 state

Algorithm

Line 1 Insert terminators in all the input transitions and

Line 2 Construct the AS_0 state (again).

insert-spec-term(*stspec*) $\triangleq$ (3.7.1.2)

```
1  cases stspec:
2   (mk-Inputspec0(vars, enab, trans)
3      → mk-Inputspec0(vars, enab, insert-trans-term(trans)),
4    mk-Contspec0(expr, prio, trans)
5      → mk-Contspec0(expr, prio, insert-trans-term(trans)),
6    mk-Priinput0(vars, trans)
7      → mk-Priinput0(vars, insert-trans-term(trans)),
8    T → stspec)
```

type: *Statespec*$_0$ → *Statespec*$_0$

Objective Insert terminators in answers in decisions and options contained in an input transition

Parameters

stspec The input transition

Result The modified AS_0 input transition

Algorithm

Line 2 If the input transition is an input node then insert terminators in the transition following the input node (*trans*)

Line 4 If the input transition is a continuous signal then insert terminators in the transition following the continuous signal (*trans*).

Line 6 If the input transition is a priority input then insert terminators in the transition following the priority input (*trans*).

Line 8 Otherwise (in the case of a save node), return the transition unchanged

insert-trans-term(mk-*Transition*$_0$(*actl*, *term*)) $\triangleq$ (3.7.1.3)

```
 1  (if actl = ⟨⟩ then
 2     mk-Transition0(⟨⟩, term)
 3    else
 4     (let mk-Actstmt0(l, act) = hd actl in
 5      if is-Decision0(act) ∨ is-Option0(act) then
 6        (let (answerl, els) = cases act:
 7                                (mk-Decision0(, ansl, els)
 8                                   → (ansl, els),
 9                                 mk-Option0(, ansl, els)
10                                   → (ansl, els)) in
11         let (actl', answerl', els') = insert-answer-term(tl actl, term, answerl, els) in
12         let act' = cases act:
13                      (mk-Decision0(q, , )
14                         → mk-Actstmt0(l, mk-Decision0(q, answerl', els')),
15                       mk-Option0(q, , )
16                         → mk-Actstmt0(l, mk-Option0(q, answerl', els'))) in
17         let mk-Transition0(actl'', t) = insert-trans-term(mk-Transition0(actl', term)) in
18         mk-Transition0(⟨act'⟩ ⌢ actl'', t))
19        else
20         if tl actl = ⟨⟩ ∧ term = nil then
21           exit("§2.6.7.1: Terminator missing in transition")
22          else
23           (let mk-Transition0(actl'', t) =
24                insert-trans-term(mk-Transition0(tl actl, term)) in
25            mk-Transition0(⟨hd actl⟩ ⌢ actl'', t))))
```

type : *Transition*$_0$ → *Transition*$_0$

Objective Insert terminators in answers of decision and option nodes in an action list

Parameters The transition containing

actl	The action list
term	The optional terminator following the action list

Result The modified AS$_0$ transition

Algorithm

Line 1	When through, return the transition containing an empty action list (the function is recursive).
Line 4	Decompose the first action statement.
Line 5	If the action is a decision or an option then
Line 6-9	Extract the answer list (*answerl*) and the else part (*els*) from the decision or option.
Line 11	Insert terminators in the answer list and in the else part. If a terminator is inserted (a join) then the rest of the action list (**tl** *actl*) is modified (*actl'*) to include a label.
Line 12-15	Construct the modified decision or option node.
Line 17	Insert terminators in the rest of the action list.
Line 18	Return the transition containing the modified decision or option (*act'*), the rest of the action list (*actl''*) and the terminator.
Line 20-25	If the first action is not a decision or option node then
Line 20-21	If the action is the last action and there is no terminator then it is an error.
Line 23-25	Insert terminators in the rest of the action list and compose the modified transition

insert-answer-term(*actl*, *term*, *answerl*, *els*) $\triangleq$ (3.7.1.4)

```
 1  (if answerl = ⟨⟩ then
 2     if els = nil then
 3       (actl, ⟨⟩, nil)
 4      else
 5       (let mk-Elsepart0(trans) = els in
 6        let (actl', trans') =
 7             cases trans:
 8              (nil
 9                 → (let (acl, term') = insert-join(actl, term) in
10                      (acl, mk-Transition0(⟨⟩, term'))),
11               mk-Transition0(actl'', term'')
12                 → if term'' = nil then
13                      (let (actl''', term''') = insert-join(actl, term) in
14                       (actl''', insert-trans-term(mk-Transition0(actl'', term'''))))
15                     else
16                      (actl, insert-trans-term(trans))) in
17        (actl', ⟨⟩, mk-Elsepart0(trans')))
18    else
19     (let (actl', answerrest, els') = insert-answer-term(actl, term, tl answerl, els) in
20      let mk-Answer0(vset, trans) = hd answerl in
21      let (actlist, trans'') =
22           cases trans:
23            (nil
24               → (let (acl, term) = insert-join(actl', term) in
25                    (acl, mk-Transition0(⟨⟩, term))),
26             mk-Transition0(actl'', term'')
27               → if term'' = nil then
28                    (let (actl''', term''') = insert-join(actl', term) in
29                     (actl''', insert-trans-term(mk-Transition0(actl'', term'''))))
30                   else
31                    (actl', insert-trans-term(trans))) in
32      (actlist, ⟨mk-Answer0(vset, trans'')⟩ ⌢ answerrest, els')))
```

type : $Actstmt_0^*$ [$Termstmt_0$] $Answer_0^*$ [$Elsepart_0$] $\rightarrow$ $Actstmt_0^*$ $Answer_0^*$ [$Elsepart_0$]

Objective Insert terminators in the answers of a decision or option node

Parameters

actl	The action list which follows the decision or option node
term	The terminator which follows the decision or option node and which follows the action list
answerl	The list of answers
els	The optional else part

Result

- The action list which follows the decision or option node. This action list is modified to include a label if it is non-empty and if any terminators is missing in the answers and in the elsepart
- The list of answers, all containing terminators
- The else part containing a terminator (unless the else part is omitted)

Algorithm

Line 1-17	When through the answer list then modify the else part.
Line 2	If the else part is omitted then return the action list, the empty list of answers and no else part.
Line 5	If the else part is specified then decompose it.

Line 6-17	Let *actl'* denote the new action list which follows the decision or option node and let *trans'* denote the new transition in the else part which contains a terminator.
Line 8	If no transition is specified then construct a join (*term*) and insert a label in the action list following the decision or option node (*acl*). The transition of the else part is an empty action list and a join (line 10).
Line 11-14	If a transition is specified in the else part, but the terminator is omitted then insert a label in the action list (*actl'''*), following the decision or option, construct a join (*term'''*), and insert terminators in the action list in the transition (line 14).
Line 16	If a transition containing a terminator is specified then insert terminator in the action list of the transition and do not modify the action list (*actl*) following the decision or option.
Line 17	Return the modified action list, an empty answer list and the modified else part.
Line 19	Insert terminators in the rest of the answers.
Line 20	Decompose the first answer in the list.
Line 21-31	Do the same for the first answer in the answer list as done for the else part as defined at line 6-16.
Line 32	Return the new action list which follows the decision or option, the first of the modified answers joined with the rest of the modified answers and the modified else part (if specified)

insert-join(*actl*, *term*) $\triangleq$ (3.7.1.5)

```
 1  (if actl = ⟨⟩ then
 2     if term = nil then
 3       exit("§2.6.7.1: Terminator missing in transition")
 4      else
 5       (let mk-Termstmt0(, t) = term in
 6        (⟨⟩, mk-Termstmt0(nil, t)))
 7    else
 8     (let mk-Actstmt0(l, act) = hd actl in
 9      let l' = if l = nil then create-unique-name() else l in
10      (⟨mk-Actstmt0(l', act)⟩ ⌢ tl actl, mk-Termstmt0(nil, mk-Join0(l')))))
```

type : $Actstmt_0^*$ [$Termstmt_0$] $\rightarrow$ $Actstmt_0^*$ $Termstmt_0$

Objective

From an action list and the following terminator, construct a join to the first action in the action list- or if it is empty, to the terminator. The join is used as terminator in the answers which are not terminating decisions (see *insert-answer-term*)

Parameters

actl	The given action list
term	The terminator which follows the action list

Result

The action list where the first action have the same label as the constructed join and the constructed join

Algorithm

Line 1	If the action list is empty then
Line 2-6	A terminator must be present and if so then
Line 5-6	Return the same (but unlabeled) terminator as specified in the transition following the decision or option.
Line 8	Decompose the first action in the action list following the enclosing decision or option.

Line 9	Extract the label from the action and if it contains no label then create one.
Line 10	Return the new action list which contains the label and return the terminator statement containing a join to that label

3.7.2 Building of Labeldict

build-state-labeldict(*statebody*)(*labeldict*) $\triangleq$ (3.7.2.1)

```
1  (if statebody = ⟨⟩ then
2     labeldict
3   else
4     (let mk-Statebody0(, specl, ) = hd statebody in
5      let specl' = ⟨specl[i] | 1 ≤ i ≤ len specl ∧ ¬is-Savespec0(specl[i])⟩ in
6      let labeldict' = build-spec-labeldict(specl')(labeldict) in
7      build-state-labeldict(tl statebody)(labeldict')))
```

type: $Statebody_0^* \rightarrow Labeldict \rightarrow Labeldict$

Objective From the state body of a process, procedure or service, collect the labels and their associated transitions to form a *Labeldict* (see the definition of *Labeldict*)

Parameters

statebody	The list of states
labeldict	The labels collected so far (the function is recursive)

Result The constructed *Labeldict*

Algorithm

Line 1	When through, return the constructed labeldict.
Line 4	Decompose the first state in the list
Line 5	Extract those input transitions which are not saves.
Line 6	Update *labeldict* to include the labels in these input transitions.
Line 7	Collect the labels in the rest of the states

build-spec-labeldict(*speclist*)(*labeldict*) $\triangleq$ (3.7.2.2)

```
1  (if speclist = ⟨⟩ then
2     labeldict
3   else
4     (let trans = s-Transition0(hd speclist) in
5      let labeldict' = build-trans-labeldict(trans)(labeldict) in
6      build-spec-labeldict(tl speclist)(labeldict')))
```

type: $Statespec_0^* \rightarrow Labeldict \rightarrow Labeldict$

Objective From the transitions attached to a state (input transitions), collect the labels and their associated transitions to form a *Labeldict* (see the definition of *Labeldict*)

Parameters

speclist	The list of input transitions
labeldict	The labels collected so far (the function is recursive)

Result The constructed *Labeldict*

Algorithm

Line 1	When through, return the constructed *Labeldict.*
Line 4-5	Update *labeldict* to include the labels in the transition of the first input transition.
Line 6	Collect the labels for the rest of the input transitions

build-trans-labeldict(**mk-***Transition*$_0$(*actl*, *term*))(*labeldict*) $\triangleq$ (3.7.2.3)

```
 1  (if actl = ⟨⟩ then
 2     (if term = nil then
 3        labeldict
 4       else
 5        (let mk-Termstmt0(l, ) = term in
 6         if l = nil then
 7            labeldict
 8           else
 9            if l ∈ dom labeldict then
10               exit("§2.6.6: Labels are not distinct")
11              else
12               labeldict + [l ↦ mk-Transition0(⟨⟩, term)]))
13    else
14     (let mk-Actstmt0(l, a) = hd actl in
15      let labeldict' =
16          if l = nil then
17            labeldict
18           else
19            if l ∈ dom labeldict then
20               exit("§2.6.6: Labels are not distinct")
21              else
22               labeldict + [l ↦ mk-Transition0(actl, term)] in
23      let labeldict'' =
24          cases a:
25           (mk-Decision0(, ansl, els)
26               → build-answerlist-labeldict(ansl, els)(labeldict'),
27            mk-Option0(, ansl, els)
28               → build-answerlist-labeldict(ansl, els)(labeldict'),
29            T → labeldict') in
30      build-trans-labeldict(mk-Transition0(tl actl, term))(labeldict'')))
```

type : *Transition*$_0$ → *Labeldict* → *Labeldict*

Objective From a transition, collect the labels and their associated transitions to form a *Labeldict* (see the definition of *Labeldict*)

Parameters A transition containing

actl	A list of action statements
term	A terminator statement
labeldict	The labels collected so far

Result The constructed *Labeldict*

Algorithm

Line 1-12	When through the list of action statements, consider the terminator
Line 5	Decompose the terminator statement in order to extract the label (*l*).
Line 6	If the label is omitted then return the constructed labeldict.

Line 9	If the label already is present in the *Labeldict* then the label has been used twice.
Line 12	Return the *Labeldict* which is updated to include the label in the terminator statement. The $Transition_0$ which follows the label has an empty list of action statements and a terminator statement
Line 14	Decompose the first action statement in the list.
Line 15-22	Let *labeldict'* denote the labeldict which is updated to include the label of the first action statement (if it is present and if the label is distinct).
Line 23-29	Let *labeldict''* denote the labeldict which is further updated to include the labels of the answers in the case of a decision or option action.
Line 30	Collect the labels in the rest of the action statements

build-answerlist-labeldict(*answerl*, *els*)(*labeldict*) $\triangleq$ (3.7.2.4)

```
 1  (if answerl = ⟨⟩ then
 2     if els = nil then
 3        labeldict
 4        else
 5        (let mk-Elsepart0(trans) = els in
 6         build-trans-labeldict(trans)(labeldict))
 7     else
 8      (let mk-Answer0(, trans) = hd answerl in
 9       let labeldict' = build-trans-labeldict(trans)(labeldict) in
10       build-answerlist-labeldict(tl answerl, els)(labeldict')))
```

type: $Answer_0^*$ [$Elsepart_0$] $\rightarrow$ *Labeldict* $\rightarrow$ *Labeldict*

Objective From the answers in a decision or option action, collect the labels and their associated transitions to form a *Labeldict* (see the definition of *Labeldict*)

Parameters

answerl	The list of answers
els	The optional else part
labeldict	The labels collected so far

Result The constructed *Labeldict*

Algorithm

Line 2-6	When through the list of answers, consider the else part.
Line 2	If the decision or option action contains no else part then return the labeldict.
Line 5	Extract the transition from the else part.
Line 6	Update *labeldict* to include the labels in the transition of the else part.
Line 8	Extract the transition of the first answer in the list.
Line 9	Update *labeldict* to include the labels of the transition.
Line 10	Collect the labels of the rest of the answers

3.7.3 Building of Statedict

build-statedict(*statelist*, *scope*)(*statedict*) $\triangleq$ (3.7.3.1)

```
1  (if statelist = ⟨⟩ then
2     statedict
3   else
4    (let mk-Statebody0(statenml, stspec, tailname) = hd statelist in
5     let speclist = ⟨(scope, stspec[i]) | 1 ≤ i ≤ len stspec⟩ in
6      cases statenml:
7      (mk-Statenamelist0(nmlist)
8         → if tailname ≠ nil ∧ elems nmlist ≠ {tailname} then
9              exit("§2.6.3: Ending name of state must be the same as starting name")
10            else
11             (let statedict' = insert-state-names(nmlist, speclist)(statedict) in
12              build-statedict(tl statelist, scope)(statedict')),
13       mk-Starredlist0(nmlist)
14         → if tailname ≠ nil then
15              exit("§2.6.3: Ending state name is not allowed in asterisk state")
16            else
17             (let totaldict = build-statedict(tl statelist, scope)(statedict) in
18              insert-starred(nmlist, speclist)(totaldict)))))
```

type : *Statebody*$_0^*$ *Qual* → *Statedict* → *Statedict*

Objective From the state body of a process, procedure or service, collect the states and their associated transitions to form a *Statedict* (see the definition of *Statedict*)

Parameters

statebody	The list of states
scope	The *Qual* denoting the scopeunit which contains the states. Every transition in *Statedict* must contain information about the context in which the transition is elaborated (since the *Statedicts* of services are merged into one *Statedict* before they are transformed into AS_1)
statedict	The states considered so far (the function is recursive)

Result The constructed *Statedict*

Algorithm

Line 1	When through, return the constructed *Statedict.*
Line 4	Decompose the first state in the list.
Line 5	Construct the list of *Specs* for the state (see the definition of *Statedict*), that is, the list of pairs of context information and input transition for the state.
Line 7-12	If a state name list is specified in the state then
Line 9	If a tailing name is specified then the state name list must consist of the same name.
Line 11	For each of the state names in the list, add a *StateD* descriptor to *Statedict.*
Line 12	Construct *StateD* descriptors for the rest of the states.
Line 13-18	If an asterisk state name list is specified then
Line 14	A tailing name is not allowed.
Line 17	Construct the *Statedict* for the rest of the states (the complete statedict) before
Line 18	The input transitions of the asterisk state are added

insert-state-names(*namelist*, *stspec*)(*statedict*) $\triangleq$ **(3.7.3.2)**

```
1  (if namelist = ⟨⟩ then
2     statedict
3   else
4    (let nm = hd namelist in
5     let stspec' =
6         if nm ∈ dom statedict then
7           (let mk-StateD(specl, ) = statedict(nm) in
8            specl)
9          else
10          ⟨⟩ in
11    let statedict' = statedict + [nm ↦ mk-StateD(stspec ⌢ stspec', nil)] in
12    if nm ∈ elems tl namelist then
13      exit("§2.6.3: State names in state must be distinct")
14     else
15      insert-state-names(tl namelist, stspec)(statedict')))
```

type : *Statenamelist*$_0$ *Speclist* $\rightarrow$ *Statedict* $\rightarrow$ *Statedict*

Objective Update *Statedict* to include the input transitions of a state node.

Parameters

namelist	The list of state names which are specified in the state node
stspec	A list of *Specs* which contains the input transitions and which, for each state in the list, is added to *Statedict*

Result The updated *Statedict*

Algorithm

Line 1	When through the state name list, return the updated *Statedict* (the function is recursive).
Line 4	Let *nm* denote the first name in the list.
Line 5-8	If that name already has an entry in *statedict* then *stspec'* denotes the input transitions associated to the entry else *stspec'* denotes the empty list of input transitions.
Line 11	Let *statedict'* denote the *Statedict* updated to include both the input transitions which already were there (*stspec'*) and the new ones (*stspec*). **nil** in *StateD* indicates that the state is not an implicit state implied from an import expression (see the definition of *Statedict*).
Line 12-13	The state name list specified in the state must contain distinct state names.
Line 15	Consider the rest of the names in the state name list

insert-starred(*namelist*, *stspec*)(*totaldict*) $\triangleq$ **(3.7.3.3)**

```
1  (if card elems namelist ≠ len namelist ∨
2     ¬(elems namelist ⊆ dom totaldict) ∨
3     elems namelist = dom totaldict ∨
4     totaldict = [] then
5      exit("§4.4: Illegal asterisk state")
6     else
7      (let restset = dom totaldict − elems namelist in
8       let namelist' = ⟨nm | nm ∈ dom restset⟩ in
9        insert-state-names(namelist', stspec)(totaldict)))
```

type : *Statenamelist*$_0$ *Speclist* $\rightarrow$ *Statedict* $\rightarrow$ *Statedict*

Objective Update *Statedict* to include the input transition from an asterisk state

Parameters

namelist	The names mentioned in the asterisk state
stspec	The *Specs* containing the input transitions of the asterisk state.

Result The update *Statedict*

Algorithm

Line 1	The names in the state name list must be distinct and
Line 2	They must not be the only occurrences of the state name and
Line 3	They must not include all states and
Line 4	There must be at least one state which is not an asterisk state.
Line 7	Extract the set of states which are covered by the asterisk state.
Line 8	Transform the set into a list.
Line 9	Add the *Specs* to *Statedict* for the deduced list of states (in the usual way)

3.7.4 Expansion of Asterisk Input, Asterisk Save and Implicit Transitions

In this section, the asterisk from input nodes and save nodes are removed from the body of a process, procedure or service and it is checked that each state contains at most one asterisk. As enabling conditions, continuous signals and import expressions have not been removed yet, the replacement of the asterisk does not include the empty queue signal and the signals implied by import.

$remove\text{-}asterisk\text{-}input\text{-}and\text{-}save(statedict)(dict) \triangleq$ (3.7.4.1)

1 $[stnm \mapsto remove\text{-}asterisk\text{-}from\text{-}state(statedict(stnm))(dict) \mid stnm \in \mathbf{dom}\ statedict]$

type : $Statedict \to Dict \to Statedict$

Objective Remove asterisk input and asterisk save from the state bodies of a process, procedure or service

Parameters

statedict	The *Statedict* representing the state bodies

Result The modified *Statedict*

Algorithm Construct the *Statedict* (the map) where every state have its asterisk input and asterisk save removed

remove-asterisk-from-state(mk-*StateD*(*speclist*,))(*dict*) $\triangleq$ (3.7.4.2)

```
1  (let (speclist', saveset, inputset, priinput) =
2       remove-asterisk-from-spec(speclist, {}, {}, {}, false)(dict) in
3   let (inputsignals, prisignals) =
4       (let qual = process-or-service-level(dict(SCOPEUNIT)) in
5         cases dict(qual):
6          (mk-ProcessD(, sig, , )      → (sig, {}),
7           mk-ServiceD(, , , sig, psig) → (sig, psig))) in
8   if priinput = prisignals ∧ (saveset ∪ inputset) ⊆ inputsignals then
9     (let (expimpinput, ) = import-export-signals(dict) in
10     let implicitinput = inputsignals \ saveset \ priinput \ inputset \ expimpinput in
11     let implicittrans = as0-implicit-transitions(implicitinput)(dict) in
12     mk-StateD(speclist' ⌢ implicittrans, nil))
13    else
14     exit("§2.6.3: Signals in state node are not well-formed"))
```

type: *StateD* → *Dict* → *StateD*

Objective Remove asterisk input and asterisk save from a state and include the implicit transitions for the unspecified signals in the *Speclist*

Parameters The *Statedict* descriptor (*StateD*) containing

speclist The *Speclist* containing the $Statespec_0$s of the state

Result The modified *StateD* descriptor

Algorithm

Line 1 Construct the modified *Speclist* and extract the signals specified in the save, the signals specified in the inputs and the signals specified in the priority inputs

Line 3-7 Extract the complete valid input signal set and the priority signals for the process or service.

Line 8 The signals in the priority inputs of the state must be equal to the priority signals for the service and the signals specified in the save and in the other inputs must be included in the complete valid input signal set.

Line 9 Let *expimpinput* denote the implicit xtQUERY and xtREPLY signals received in export and import shorthands and the emptyq signal.

Line 10 The signals for which implicit transitions are to be constructed are the signals in the complete valid input signal set minus the signals mentioned in the save minus the high priority signals minus the signals mentioned in the input minus the implicit signals.

Line 11 Construct the implicit transitions.

Line 12 Construct the modified *StateD* descriptor

as_0-*implicit-transitions*(*signalqualset*)(*dict*) $\triangleq$ (3.7.4.3)

```
1  if signalqualset = {} then
2    ⟨⟩
3   else
4    (let signalqual ∈ signalqualset in
5     let inputvars = as0-inputvars({signalqual})(dict) in
6     let input = mk-Inputspec0(inputvars, nil,
7                               mk-Transition0(⟨⟩, mk-Termstmt0(nil, mk-Nextstate0(nil)))) in
8     ⟨(dict(SCOPEUNIT)), input⟩ ⌢ as0-implicit-transitions(signalqualset \ {signalqual})(dict))
```

type: *Signalqual*-**set** → *Dict* → *Speclist*

Objective Construct the implicit transitions for a state

Parameters

signalqualset The set of signals for which implicit transitions are to be constructed

Result A *Speclist* containing the implicit transitions

Algorithm

Line 1 When through, return the input *Speclist.*

Line 4 Let *signalqual* denote the next signal to be dealt with.

Line 5 Construct the parameter list for the signal.

Line 6 Construct the input transition for the signal. The transition contains a nextstate dash only.

Line 8 Construct a *Spec* and join it with the *Speclist* for the rest of the signals

remove-asterisk-from-spec(*speclist*, *sigset*, *saveset*, *priset*, *asterisk*)(*dict*) $\triangleq$ (3.7.4.4)

```
(if speclist = ⟨⟩ then
   (⟨⟩, sigset, saveset, priset)
  else
   (let (q, spec) = hd speclist in
    cases spec:
     (mk-Savespec0(sigl)
        → if is-Starred0(sigl) then
            if asterisk then
              exit("§4.7: State has more than one asterisk input or save")
             else
              (let (specrest, inputtot, savetot, pritot) =
                   remove-asterisk-from-spec(tl speclist, sigset, saveset, priset, true)(dict) in
               let sigq = all-input-signals(dict(SCOPEUNIT))(dict) in
               let (expimpinput, ) = import-export-signals(dict) in
               let sigq' = sigq \ inputtot \ savetot \ pritot \ expimpinput in
               let as0siglist = ⟨as0-id(sig) | sig ∈ sigq'⟩ in
               let spec' = if as0siglist = ⟨⟩ then
                                ⟨⟩
                           else
                                ⟨(q, mk-Savespec0(as0siglist))⟩ in
               (specrest ⌢ spec', inputtot, sigq' ∪ savetot, pritot))
          else
           (let sigq = transform-signallist(sigl)(dict) in
            let (specrest, inputtot, savetot, pritot) =
                remove-asterisk-from-spec(tl speclist, sigset, saveset ∪ sigq, priset, asterisk)(dict) in
            if sigq ∩ (sigset ∪ saveset ∪ priset) = {} then
              (specrest ⌢ ⟨hd speclist⟩, inputtot, savetot, pritot)
             else
              exit("§2.6.3: Signals in state are not disjoint")),
```

```
30      mk-Priinput0(inp, ),
31      mk-Inputspec0(inp, enab, trans)
32        → if is-Starred0(inp) then
33            if asterisk then
34              exit("§4.7: State has more than one asterisk input or save")
35             else
36              (let (specrest, inputtot, savetot, pritot) =
37                   remove-asterisk-from-spec(tl speclist, sigset, saveset, pritot, true)(dict) in
38               let sigq = all-input-signals(dict(SCOPEUNIT))(dict) in
39               let (expimpinput, ) = import-export-signals(dict) in
40               let sigq' = sigq \ inputtot \ savetot \ pritot \ expimpinput in
41               let inpvars = as0-inputvars(sigq')(dict) in
42               let spec' = if inpvars = ⟨⟩ then
43                                ⟨⟩
44                           else
45                                ⟨(q, mk-Inputspec0(inpvars, enab, trans))⟩ in
46               (specrest ⌢ spec', sigq' ∪ inputtot, savetot, pritot))
47           else
48            (let sigq = {signal-qual(id)(dict) | mk-Inputvars0(id, ) ∈ elems inp} in
49             let (sigset', priset') = if is-Priinput0(spec) then
50                                          (sigset, priset ∪ sigq)
51                                      else
52                                          (sigset ∪ sigq, priset) in
53             let (specrest, inputtot, savetot, pritot) =
54                 remove-asterisk-from-spec(tl speclist, sigset', saveset, priset', asterisk)(dict) in
55             if sigq ∩ (sigset ∪ saveset ∪ priset) = {} then
56               (specrest ⌢ ⟨hd speclist⟩, inputtot, savetot, pritot)
57              else
58               exit("§2.6.3: Signals in state are not disjoint")),
59      T → remove-asterisk-from-spec(tl speclist, sigset, saveset, priset, asterisk)(dict))))
```

type : $Statespec_0^*$ *Signalqual*-set *Signalqual*-set *Signalqual*-set *Bool* → *Dict* →
$Statespec_0^*$ *Signalqual*-set *Signalqual*-set *Signalqual*-set

Objective

Remove asterisk input and asterisk save from a state. In addition to the input nodes, the function also returns the set of signals mentioned in the input nodes, the set of signals mentioned in the save nodes and the set of signals mentioned in the priority input nodes

Parameters

speclist	The list of remaining inputs and saves to be dealt with (the function is recursive)
sigset	The *Quals* of the signals which have been recognized in an input node so far
saveset	The *Quals* of the signals which have been recognized in an save node so far
priset	The *Quals* of the signals which have been recognized in an priority input node so far
asterisk	A flag indicating whether an asterisk save or asterisk input has been recognized so far

Result

The modified list of inputs and saves constructed so far, the complete *Qual* set of input signals, the complete *Qual* set of save signals and the set of high priority signals received in the state,

Algorithm

Line 1	When through, return the complete *Qual* set of input signals, the complete *Qual* set of save signals and the set of high priority signals
Line 4	Decompose the first *Spec* of the state.

Line 6-27	If the *Statespec*$_0$ in the first *Spec* is a save node then
Line 6	If an asterisk is specified in the save node then
Line 8	If an asterisk save or asterisk input already has been encounted then it is an error.
Line 11	Remove the asterisk input and save from the rest of the input and save nodes. **true** indicates that an asterisk save (or input) now have been encounted. *inputtot* and *savetot* are the complete sets of input signals and save signals explicitly specified.
Line 13	Extract the *Quals* of the signals which are contained in the complete valid input signal set for the service or process.
Line 14-15	The set of signals to be specified in the save node are the signals in the complete valid input signal set (*sigq*) minus the signals specified in the input nodes *inputtot* minus the signals specified in the other save nodes (*savetot*) minus the priority input signals minus the signals implied from import/export.
Line 16	Convert the set of signal *Quals* to an AS$_0$ list of identifiers.
Line 17	Construct the new save *Spec* provided there exist any signals to save.
Line 21	Return the *Spec* joined with the rest of the *Specs*, the complete set of input signals, the complete set of save signals and the set of high priority signals.
Line 23	If the save contains a signallist then transform the signallist to obtain the signals specified (*sigq*).
Line 24	Remove the asterisk input and save from the rest of the input and save nodes where the set of save signals obtained so far is updated to include the signals of this save node.
Line 26	The signals in the save must be disjoint from the other signals in the state.
Line 27	Add the *Spec* to the resulting list of *Specs* (and leave it unchanged).
Line 31-59	If the *Statespec*$_0$ in the first *Spec* is an input node then
Line 32	If an asterisk is specified in the input node then
Line 33	If an asterisk save or asterisk input already has been encounted then it is an error.
Line 36	Remove the asterisk input and save from the rest of the input and save nodes. **true** indicates that an asterisk input (or save) now have been encounted.
Line 38	Extract the *Quals* of the signals which are contained in the complete valid input signal set for the service or process.
Line 39-40	The set of signals to be specified in the input node are the signals in the complete valid input signal set (*sigq*) minus the signals specified in the other input nodes *inputtot* minus the signals specified in the save nodes (*savetot*) minus the priority input signals minus the signals implied from import/export.
Line 41	Convert the set of signal *Quals* to a list of *Inputvars*$_0$s where the variables to receive the values are omitted (**nil**).
Line 42	Construct the new input *Spec* provided there are any signals not explicitly mentioned.
Line 46	Return the *Spec* joined with the rest of the *Specs*, the complete set of input signals and the complete set of save signals.
Line 48	If the input contains a list of *Inputvars*$_0$s then extract the *Qual* of the signals contained in them
Line 49	if the input is a priority input then add the signals to the priority input set (*priset*) else add the signals to the set of normal signals (*sigset*).
Line 53	Remove the asterisk input and save from the rest of the input and save nodes

Line 55	The signals in the input must be disjoint from the other signals in the state.
Line 56	Add the *Spec* to the resulting list of *Specs* (and leave it unchanged).
Line 59	If the $Statespec_0$ is a continuous signal, then continue with the next $Statespec_0$

$as_0\text{-}inputvars(starinput)(dict) \triangleq$ (3.7.4.5)

```
1  (if starinput = {} then
2     ⟨⟩
3    else
4     (let qual ∈ starinput in
5      let (nm, ) = qual[len qual] in
6      let id = mk-Id0(⟨⟩, nm) in
7      let t = cases dict(qual):
8              (mk-SignalD(tli, , ) → tli,
9               mk-TimerD(tli, ) → tli) in
10     let vnum = len t in
11     let inpu = mk-Inputvars0(id, ⟨nil | 1 ≤ i ≤ vnum⟩) in
12     ⟨inpu⟩ ⌢ as0-inputvars(starinput \ {qual})(dict)))
```

type : $Signalqual\text{-}\mathbf{set} \rightarrow Dict \rightarrow Inputvars_0^*$

Objective Construct the AS_0 variable lists for the signals which are received in an asterisk input node or received in implicit transitions.

Parameters

starinput	The set of valid input signals for a process which neither have been specified explicitly in an input node, priority inputnode nor in a save node

Algorithm

Line 1	When through, return nothing (the function is recursive).
Line 4	Let *qual* denote a signal *Qual* in the set.
Line 5	Extract the name of the signal.
Line 6	Construct the identifier of the signal.
Line 7-9	Let *t* denote the sort list of the signal or timer.
Line 10	Let vnum denote the number of values conveyed by the signal.
Line 11	Construct the AS_0 input variable list where every variable is **nil** (omitted) and
Line 12	Return it together with those for the rest of the signals

3.7.5 Expansion of Continuous Signals and Enabling Condition

remove-cont-enable-from-statelist(*stateset*)(*dict*, *statedict*) $\triangleq$ (3.7.5.1)

```
1  (if stateset = {} then
2     statedict
3    else
4     (let state ∈ stateset in
5      let mk-StateD(speclist, ) = statedict(state) in
6      let statedict' = remove-cont-enable-from-state(speclist, state)(dict, statedict) in
7      remove-cont-enable-from-statelist(stateset \ {state})(dict, statedict')))
```

type : $Statename_0\text{-}\mathbf{set} \rightarrow Dict\ Statedict \rightarrow Statedict$

Objective Remove all continuous signal and enabling condition from *Statedict* by replacing the appropriate *StateD* descriptors by *ContenablestateD* descriptors and by adding new *StateD* descriptors for the implied states

Parameters

stateset The names of the states which shall have the continuous signals and enabling conditions removed. Initially this set includes all states (the function is recursive)

Result The modified *Statedict*

Algorithm

Line 1 When through, return the modified *Statedict.*

Line 4-5 Let *speclist* denote the *Speclist* of one of the remaining states.

Line 6 Remove continuous signals and enabling conditions in that state.

Line 7 Remove continuous signals and enabling conditions in the rest of the states

remove-cont-enable-from-state(*speclist*, *stnm*)(*dict*, *statedict*) $\triangleq$ (3.7.5.2)

```
1  (let enable = (∃(, sp) ∈ elems speclist)(is-Inputspec0(sp) ∧ s-Enabling0(sp) ≠ nil) in
2   let cont = (∃(, sp) ∈ elems speclist)(is-Contspec0(sp)) in
3   if ¬enable ∧ ¬cont then
4     statedict
5    else
6     (let (emptyqid, formu1nm, formu2nm) = dict(GLOBALNAMES) in
7      let formu1 = mk-Id0(⟨⟩, formu1nm),
8          formu2 = mk-Id0(⟨⟩, formu2nm) in
9      let ptrans = pretrans(emptyqid, formu1) in
10     let speclist' =
11         if cont
12           then expand-continuous(stnm, speclist, emptyqid, formu1, formu2)(dict)
13           else speclist in
14     if ¬enable then
15       (let newstnm = create-unique-name() in
16        let preterm = mk-Transition0(ptrans, mk-Nextstate0(newstnm)) in
17        statedict + [stnm    ↦ mk-ContenablestateD(preterm),
18                     newstnm ↦ mk-StateD(speclist', nil)])
19      else
20       (let emptyqinput =
21            if cont then
22              ⟨⟩
23             else
24              (let term = mk-Termstmt0(nil, mk-Nextstate0(stnm)) in
25               ⟨emptyqtrans(term, emptyqid, formu1, formu2)(dict)⟩) in
26        let (mk-Transition0(decision, ), statedict') =
27            expand-enable(speclist' ⌢ emptyqinput)(statedict) in
28        let preterm = mk-Transition0(ptrans ⌢ ⟨decision⟩, nil) in
29        statedict' + [stnm ↦ mk-ContenablestateD(preterm)])))
```

type: *Speclist Name*$_0$ $\rightarrow$ *Dict Statedict* $\rightarrow$ *Statedict*

Objective Remove continuous signals and enabling conditions from a state by modifying *Statedict*

Parameters

speclist The *Speclist* containing the input transitions of the state

stnm The name of the state

Result The modified *Statedict*

Algorithm

Line 8-10	If the priority is omitted in a continuous signal then no other continuous signals may be specified.
Line 1-4	If the state neither contains enabling conditions nor continuous signals then leave the *Statedict* unchanged.
Line 6	Extract the AS_0 identifier of the empty queue signal and the AS_0 names of the two variables used in connection with the empty queue signals.
Line 7-8	Construct AS_0 identifiers of the two variables.
Line 9	Construct the AS_0 action list which updates the "unique" variable (*formu1*) and which outputs the empty queue signal.
Line 10-13	Expand continuous signals by modifying the *Speclist* (*speclist*).
Line 14	If the state contains no enabling conditions then
Line 15	Create a new name for the state (which contains continuous signals).
Line 16	Construct the transition which must be interpreted in every next-state containing this state.
Line 17	Update *Statedict* with two descriptors: The first is used in the transformation of nextstate nodes. When a nextstate node contains the name *stnm* then the nextstate node is replaced by *preterm*. The second one is used when the state containing continuous signals (*stnm*) is transformed into AS_1.
Line 20-29	Expand enabling conditions.
Line 20-25	Construct the empty queue input unless it already is included in *speclist'*, i.e. unless the state contains continuous signals. When a state contains no continuous signals, the transition following the empty queue is considerable simpler since no decision nodes are present in that case (only a nextstate back to the same state is implied).
Line 26	Expand the enabling condition. The result of this expansion is a composite decision action (*decision*) testing on the expressions in the enabling condition and the *Statedict* which is extended to include *StateD* descriptors of all the implicit states involved.
Line 28	Construct the transition to be interpreted in the appropriate next-state nodes (as in line 16-17) and
Line 29	Replace the *StateD* descriptor for the state, by a *ContenablestateD* in *Statedict* (as in line 16-17)

$expand\text{-}continuous(statenm, speclist, emptyqid, formu1, formu2)(dict) \triangleq$ (3.7.5.3)

```
 1  if (∃(,stsp) ∈ elems speclist)(is-Contspec0(stsp) ∧ s-Priority0(stsp) = nil) ∧
 2    card {(,stsp) ∈ elems speclist | is-Contspec0(stsp)} > 1 then
 3     exit("§4.11: Only one continuous signal may be specified if the priority is omitted")
 4    else
 5     (let priomap = collect-priority-info(speclist, len speclist)(dict) in
 6      let conttrans = build-cont-trans(speclist, statenm, priomap) in
 7      let contterm = mk-Transition0(conttrans, nil) in
 8      let normtrans = ⟨speclist[i] | i ∈ ind speclist ∧ ¬is-Contspec0(speclist[i])⟩ in
 9      let emptyqinput = emptyqtrans(contterm, emptyqid, formu1, formu2)(dict) in
10      normtrans ⌢ ⟨emptyqinput⟩)
```

type : $Statename_0\ Speclist\ Sigid_0\ Varid_0\ Varid_0 \rightarrow Dict \rightarrow Speclist$

Objective Modify the input transition for a state containing continuous signals.

Parameters

statenm	The name of the state containing continuous signals
speclist	The list of *Specs* which contains the input transitions
emptyqid	The AS_0 identifier of the empty queue signal
formu1,formu2	The AS_0 identifiers of the variables used in the continuous signal model

Result The modified list of *Specs*

Algorithm

Line 1	There
Line 5	Construct a map which contains the relation between the priority of a signal, and its position (index) in *speclist.*
Line 6-7	Construct the composite decision action which tests on the various expression in the order given by the priorities.
Line 8	Remove the input transitions, which contains continuous signals, from *speclist.*
Line 9	Construct the empty queue input transition which tests on whether it is the right empty queue signal (i.e. test whether *formu1* equals *formu2*) and which uses the composite decision (*contterm*) in the "true" branch.
Line 10	Add the empty queue input transition to the input transitions containing no continuous signals

pretrans(*emptyqid*, *formu1id*) $\triangleq$ (3.7.5.4)

```
1  (let lit1 = mk-Name0("1", nil),
2       self = mk-Selfexpr0() in
3   let infix = mk-Infixexpr0(formu1id, PLUS, lit1) in
4   let update = mk-Assignstmt0(formu1id, infix) in
5   let task = mk-Task0(⟨update⟩),
6       outtoself = mk-Output0(⟨mk-Outputsig0(emptyqid, ⟨formu1id⟩)⟩, self, ⟨⟩) in
7   ⟨mk-Actstmt0(nil, task), mk-Actstmt0(nil, outtoself)⟩)
```

type: $Sigid_0\ Varid_0 \rightarrow Actstmt_0^+$

Objective Construct the action statements:

```
TASK    formu1id := formu1id + 1;
OUTPUT emptyqid(formu1id) TO SELF;
```

Algorithm

Line 1	Construct the integer literal "1".
Line 2	Construct the PiD self expression.
Line 3	Construct `formu1id + 1`.
Line 4	Construct `formu1id := formu1id + 1`.
Line 5	Construct `TASK formu1id := formu1id + 1`.
Line 6	Construct `OUTPUT emptyqid(formu1id) TO SELF`.
Line 7	Return the two actions as action statements (without labels)

emptyqtrans(*conttrans*, *emptyqid*, *formu1id*, *formu2id*)(*dict*) $\triangleq$ (3.7.5.5)

```
1  (let emptyqinput = mk-Inputvars0(emptyqid, ⟨formu2id⟩) in
2   let ans1 = mk-Answer0(⟨mk-Condition0(nil, mk-Name0("TRUE", nil))⟩, conttrans),
3       ans2 = mk-Answer0(⟨mk-Condition0(nil, mk-Name0("FALSE", nil))⟩,
4                         mk-Termstmt0(nil, mk-Nextstate0(nil))) in
5   let decision = mk-Decision0(mk-Infixexpr0(formu1id, EQ, formu2id),
6                               ⟨ans1, ans2⟩, nil) in
7   let emptytrans = mk-Transition0(⟨mk-Actstmt0(nil, decision)⟩, nil) in
8   (dict(SCOPEUNIT), mk-Inputspec0(⟨emptyqinput⟩, nil, emptytrans)))
```

type : *Transition*$_0$ *Sigid*$_0$ *Varid*$_0$ *Varid*$_0$ $\rightarrow$ *Dict* $\rightarrow$ *Spec*

Objective Construct the input transition:

```
INPUT emptyqid(formu2id);
 DECISION (formu1id = formu2id)
  (true)  : conttrans;
  (false) : NEXTSTATE(-);
 ENDDECISION;
```

and return the *Spec* which contains the input transition

Parameters

conttrans	The transition which correspond to the "true" branch. If the state contains no continuous signals (but contains enabling conditions) then it contains a nextstate to the original state, otherwise it contains the composite decision constructed in *build-cont-trans*. The nextstate in the "false" branch implies continuation at the implicit state (not the original state) to which the input transition is associated
emptyqid	The AS$_0$ identifier of the empty queue signal
formu1id,formu2id	The AS$_0$ identifiers of the variables used in continuous signal and enabling condition

build-cont-trans(*speclist*, *stnm*, *priomap*) $\triangleq$ (3.7.5.6)

```
1  (let i ∈ dom priomap be s.t. ¬(∃n ∈ dom priomap)(n < i) in
2   let trans = if priomap \ {i} = []
3               then mk-Transition0(⟨⟩, mk-Termstmt0(nil, mk-Nextstate0(stnm)))
4               else build-cont-trans(speclist, stnm, priomap \ {i}) in
5   let index = priomap(i) in
6   let (scope, mk-Contspec0(expr, , exittrans)) = speclist[index] in
7   let yesval = mk-Condition0(nil, mk-Id0(⟨⟩, mk-Name0("TRUE", nil))),
8       noval = mk-Condition0(nil, mk-Id0(⟨⟩, mk-Name0("FALSE", nil))) in
9   let yestrans = mk-Answer0(⟨yesval⟩, exittrans),
10      notrans = mk-Answer0(⟨noval⟩, trans) in
11  let decision = mk-Decision0(mk-Scopeexpr0(scope, expr), ⟨yestrans, notrans⟩, nil) in
12  mk-Transition0(⟨mk-Actstmt0(nil, decision)⟩, nil))
```

type : *Speclist Statename*$_0$ $(N_1 \xrightarrow[m]{} N_1) \rightarrow$ *Transition*$_0$

Objective Construct the composite decision which tests and branch on the expressions in continuous signals in the order given by the priorities

Parameters

speclist	The *Specs* which contains continuous signals
stnm	The name of the (explicit) state to which the continuous signals are attached

priomap — The relation between the priority of continuous signals and their position (index) in *speclist*

Result — The composite decision action statement

Algorithm

Line 1 — Let i denote the entry in the priority map which has the lowest value (i.e. the highest priority among the remaining elements in *priomap*).

Line 2-4 — If this value has the lowest priority (i.e. if it is the only one left in the map) then let *trans* denote the terminator statement containing a nextstate to the old explicit state (line 3) else let *trans* denote the decision action for the rest of the priorities (line 4).

Line 5-6 — Let *scope* denote the *Qual* of the scopeunit which defined the state (recall that the *Statedicts* of services have been merged before this transformation takes place and that the evaluation of the expression depends on the actual service), let *expr* denote the expression in the continuous signal and let *exittrans* denote the transition in the continuous signal

Line 7-12 — construct the transition containing the decision:

```
DECISION (expr):
 (true)  : exittrans;
 (false) : trans;
ENDDECISION
```

where it is specified that *expr* shall be transformed into AS_1 in the context of the scopeunit denoted by *scope* which either is a process, a procedure or a service

collect-priority-info(*speclist*, *index*)(*dict*) $\triangleq$ (3.7.5.7)

```
1  (if index = 0 then
2     []
3   else
4    (let emap = collect-priority-info(speclist, index − 1)(dict) in
5     let (scope, inp) = speclist[index] in
6     let dict' = [q ↦ dict(q) | q ∈ dom dict ∧ ¬is-SynD(dict(q))] + [SCOPEUNIT ↦ scope] in
7      cases inp:
8       (mk-Contspec0(, prio, )
9          → (let prio' = if prio = nil then 1 else eval-simple-expr(prio, "INTEGER")(dict') in
10            if prio' ∈ dom emap then
11              exit("§4.11: Several continuous signals with the same priority")
12             else
13              emap + [prio' ↦ index]),
14       T → emap)))
```

type: $Speclist\ N_0 \to Dict \to (N_1 \xrightarrow{m} N_1)$

Objective — Construct a map containing the relation between the priority of a continuous signal and the index value of the *Spec* in a *Speclist* containing the continuous signal

Parameters

speclist — The *Speclist* which contains the continuous signals

index — The current index value into *speclist*. Initially, the index value equals the length of the list. The recursion terminates when the index value reaches zero

Result — The constructed priority map

Algorithm

Line 1	When through, return the empty map.
Line 4	Collect the priorities of the rest of the continuous signals.
Line 5	Extract the *Qual* of the scopeunit which defined the continuous signal and the input transition.
Line 6	Override *Dict* with the *Qual* of the scopeunit such that the priority expression can be evaluated in the right context. Also exclude synonyms from *Dict* because it is illegal to specify a synonym in the priority construct.
Line 8-13	If the input transition is a continuous signal then Determine the priority of the continuous signal. If the priority is omitted then set the priority to "1" else evaluate the simple expression.
Line 10-11	There must not exist another continuous signal which has the same value.
Line 13	Add the priority to the map containing the rest of the priorities.
Line 14	If the input transition is not a continuous signal then return the map containing the priorities of the rest of the continuous signals

expand-enable(*speclist*)(*statedict*) $\triangleq$ (3.7.5.8)

```
1  (let enabset = {speclist[i] | i ∈ ind speclist ∧ is-Inputspec0(speclist[i]) ∧
2                                  s-Enabling0(s-Statespec0(speclist[i])) ≠ nil} in
3   let normtrans = ⟨speclist[i] | 1 ≤ i ≤ len speclist ∧ speclist[i] ∉ enabset⟩ in
4   build-enable-decision(normtrans, enabset)(statedict))
```

type: *Speclist* → *Statedict* → *Transition*$_0$ *Statedict*

Objective Construct the enclosed decision actions which models an enabling condition state and update *Statedict* with the implied implicit states

Parameters

speclist A *Speclist* containing the input transitions of a state

Result A transition containing the nested decision action and the updated *Statedict*

Algorithm

Line 1-2	Let *enabset* denote the set of *Specs* which contains enabling condition and
Line 3	Let *normtrans* denote the set of *Specs* which contains no enabling condition.
Line 4	Construct the decision action and the implicit states

build-enable-decision(*normtrans*, *enabset*)(*statedict*) $\triangleq$ (3.7.5.9)

```
1  (let falsevalue = mk-Condition0(nil, mk-Id0(⟨⟩, mk-Name0("FALSE", nil))) in
2   let truevalue = mk-Condition0(nil, mk-Id0(⟨⟩, mk-Name0("TRUE", nil))) in
3   let enab ∈ enabset in
4   let (scope, mk-Inputspec0(vl, expr, tr)) = enab in
5   let sigl = ⟨sigid | (∃varl ∈ elems vl)(s-Sigid0(varl) = sigid)⟩ in
6   let newtrans = normtrans ⌢ ⟨(scope, mk-Savespec0(sigl))⟩ in
7   let newtrans' = normtrans ⌢ ⟨(scope, mk-Inputspec0(vl, nil, tr))⟩ in
8   let (trans0, statedict') =
9       if enabset \ {enab} = {}
10        then build-implicit-state(newtrans)(statedict)
11        else build-enable-decision(newtrans, enabset \ {enab})(statedict) in
12  let (trans1, statedict'') =
13      if enabset \ {enab} = {}
14        then build-implicit-state(newtrans')(statedict')
15        else build-enable-decision(newtrans', enabset \ {enab})(statedict') in
16  let ans0 = mk-Answer0(falsevalue, trans0) in
17  let ans1 = mk-Answer0(truevalue, trans1) in
18  let decexpr = mk-Scopeexpr0(scope, expr) in
19  let decision = mk-Decision0(decexpr, ⟨ans0, ans1⟩, nil) in
20  let actstmt = mk-Actstmt0(nil, decision) in
21  (mk-Transition0(⟨actstmt⟩, nil), statedict''))
```

type: *Speclist Spec*-**set** → *Statedict* → *Transition*$_0$ *Statedict*

Objective See *expand-enable*

Parameters

normtrans	A *Speclist* which contains the input transitions which does not contain an enabling condition.
enabset	A set of *Specs* containing the input transitions which contains an enabling condition.

Result See *expand-enable*

Algorithm

Line 1	Construct the "false" case label
Line 2	Construct the "true" case label
Line 3	Extract a *Spec* from the *Spec* set representing the input transitions with an enabling condition
Line 6	Decompose the *Spec*
Line 5	Extract the signal identifiers from the input node
Line 6	Form a new *Speclist* containing the old *Speclist* and a *Savespec*$_0$.
Line 7	Form a new *Speclist* containing the old *Speclist* and an *Inputspec*$_0$.
Line 8-11	Construct the decision tree for the "false" case.
Line 10	If *enab* is the last enabling condition, construct a decision leading the *Nextstate*$_0$s.
Line 11	if *enab* is not the last enabling condition, construct a decision leading to decisions.
Line 12-15	Construct the decision tree for the "true" case.
Line 16	Construct the "false" answer.
Line 17	Construct the "true" answer.
Line 18	Construct the decision expression.
Line 19	Construct the decision.
Line 20	Construct the action statement.
Line 21	Return the new transition and the updated *Statedict*.

build-implicit-state(*alltrans*)(*statedict*) $\triangleq$ (3.7.5.10)

```
1  (let stnm = create-unique-name() in
2   let trans = mk-Transition0(⟨⟩, mk-Termstmt0(nil, mk-Nextstate0(stnm))) in
3   (trans, statedict + [stnm ↦ mk-StateD(alltrans, nil)]))
```

type : *Speclist* → *Statedict* → *Transition*$_0$ *Statedict*

Objective Create a new implicit state with the complete *Speclist* and update the *Statedict*

Parameters

alltrans The complete *Speclist.*

Result See *expand-enable*

Algorithm

Line 1 Create an AS_0 name for the new implicit state.

Line 2 Construct the transition containing the nextstate which is to the implicit state.

Line 3 Return the transition and the updated *Statedict*

3.7.6 Transformation of States and Input Nodes

transform-statelist(*transformedstates*)(*dict*) $\triangleq$ (3.7.6.1)

```
1  (let statedict = dict(STATEDICT) in
2   if dom statedict = transformedstates then
3     ({}, dict)
4    else
5     (let stnm ∈ (dom statedict \ transformedstates) in
6      if is-ContenablestateD(statedict(stnm)) then
7        transform-statelist(transformedstates ∪ {stnm})(dict)
8       else
9        (let mk-StateD(speclist, importinf) = statedict(stnm) in
10        let (currentstate, impact) = if importinf = nil then (stnm, ) else importinf in
11        let (saveset1, inputset1, dict') = transform-statespecl(currentstate, speclist)(dict) in
12        let inputset1' =
13            if importinf = nil ∨ impact = nil
14              then inputset1
15              else insert-importact(impact, inputset1) in
16        let statenode = mk-State-node1(name-to-name1(stnm), saveset1, inputset1') in
17        let (bodyrest, dict'') = transform-statelist(transformedstates ∪ {stnm})(dict') in
18        ({statenode} ∪ bodyrest, dict'')))
```

type : *Statename*$_0$-**set** → *Dict* → *State-node*$_1$-**set** *Dict*

Objective Transform the states of a process or procedure body into AS_1. The states are transformed by considering every state in *Statedict.* The implicit states implied from services, continuous signals and enabling conditions are already included in the AS_0 states that are to be transformed. During the transformation, new states implied by import expressions may be added to *Statedict*

Parameters

transformedstates The AS_0 names of the states which already have been transformed. When *transform-statelist* initially is applied, the set is empty

Result The AS_1 states and a *Dict* updated with information about the implicit variables (the IMPLIED entry) implied by import expressions used in the process or procedure and updated to include the output signals (the *OUTSIGNALS* entry) used in the process or procedure.

Algorithm

Line 1 Extract the *Statedict* from *Dict*

Line 2 When the transformed states are all states in *Statedict* then return.

Line 5 Let *stnm* denote a state which has not been transformed.

Line 6-7 If the state contained continuous signals or enabling conditions then the state has been replaced by a number of implicit states and is therefore not transformed.

Line 9 Decompose the *Statedict* descriptor of the state. If *importinf* is different from **nil** then the state is an import state (see the definition of *Importstateinf*).

Line 10 Let *currentstate* denote the state name to be inserted in the next-state dash of the input transitions. If the state in hand is an implicit state implied from an import expression then the state name is found in the *StateD* descriptor.

Line 11 Transform the inputs and the save of the state.

Line 12-14 If the state is an implicit state implied from import then the AS_1 action which uses the implicit import variable must be inserted in the transformed state. This action was not inserted while transforming the import expression because it is during the transformation of expressions that the import states are constructed.

Line 16 Construct the AS_1 state node.

Line 17 Transform the rest of the states.

Line 18 Join the constructed state with the rest of the states and also return the updated *Dict*

transform-statespecl(*stnm*, *statespeclist*)(*dict*) $\triangleq$ **(3.7.6.2)**

```
1  (let (saveset, statespeclist') =
2       if (∃i ∈ ind statespeclist)(is-Savespec0(s-Statespec0(statespeclist[i]))) then
3         (let i be s.t. (is-Savespec0(s-Statespec0(statespeclist[i]))) in
4          let mk-Savespec0(sigl) = s-Statespec0(statespeclist[i]) in
5          let sigqualset = transform-signallist(sigl)(dict) in
6          (sigqualset, ⟨statespeclist[n] | 1 ≤ n ≤ len statespeclist ∧ n ≠ i⟩))
7        else
8         ({},statespeclist) in
9  let dict' = dict + [SCOPEUNIT ↦ process-level(dict(SCOPEUNIT))] in
10 let varset = {qual ∈ dom dict | is-VarD(dict(qual)) ∧ process-level(qual) = dict'(SCOPEUNIT)} in
11 let xtQUERYspec0 = as0-xtQUERY-inputs(varset, stnm)(dict) in
12 let (as1nodeset, dict') = transform-input(stnm, statespeclist' ⌢ xtQUERYspec0)(dict) in
13 let saveset1 = mk-Save-signalset1(make-as1idset(saveset)(dict)) in
14 (saveset1, as1nodeset, dict'))
```

type: *Statename*$_0$ *Speclist* $\rightarrow$ *Dict* $\rightarrow$ *Save-signalset*$_1$ *Input-node*$_1$-**set** *Dict*

Objective Transform a state into AS_1

Parameters

stnm The name of the state. It is inserted in the nextstate dash in the state

statespeclist The *Speclist* containing the input transitions of the state

Result The AS_1 save node and the AS_1 input nodes

Algorithm

Line 1-8	Let *saveset* denote the *Signalqual* set containing the signals which are saved in the state and let *statespeclist'* denote the modified *Speclist* where the save has been removed (if any).
Line 2	If there exist a save in the *Speclist* then
Line 3-4	Extract the save from the *Speclist.*
Line 5	Transform the signals in the save.
Line 6	Remove the save from the *Speclist*
Line 9-10	Update SCOPEUNIT to denote the surrounding process. *dict'* is only used for extracting the variables local to the process (line 10).
Line 11	Construct the AS_0 input transitions for the xtQUERY signals.
Line 12	Transform the input transitions where the xtQUERY inputs have been added.
Line 13	Construct the AS_1 save node from the set of *Signalquals.*
Line 14	Return the save node and the input nodes

$as_0\text{-}xtQUERY\text{-}inputs(exportset, stnm)(dict) \triangleq$ (3.7.6.3)

```
1  (if exportset = {} then
2     ⟨⟩
3   else
4    (let qual ∈ exportset,
5         term = mk-Nextstate0(stnm),
6         mk-VarD(tq, , export, , ) = dict(qual) in
7     if export = nil then
8       as0-xtQUERY-inputs(exportset \ {qual}, stnm)(dict)
9      else
10      (let (,nm) = qual[len qual],
11           mk-SignalnamesD(imp, xq, xr) = exportmap((tq, nm)) in
12       let sigelem = ⟨mk-Outputsig0(mk-Id0(⟨⟩, xr), ⟨mk-Id0(⟨⟩, imp)⟩)⟩ in
13       let xroutput = mk-Output0(sigelem, mk-Senderexpr0(), ⟨⟩) in
14       let xrtrans = mk-Transition0(⟨mk-Actstmt0(nil, xroutput)⟩, term) in
15       let xqinput = mk-Inputspec0(⟨mk-Inputvars0(mk-Id0(⟨⟩, xq), ⟨⟩)⟩, nil, xrtrans) in
16       ⟨(dict(SCOPEUNIT), xqinput)⟩ ⌢ as0-xtQUERY-inputs(exportset \ {qual}, stnm)(dict))))
```

type : *Qual*-**set** $Name_0 \rightarrow Dict \rightarrow Speclist$

Objective Create AS_0 input nodes for all the signals, implied from export, for a process as defined in §4.13 of Z.100

Parameters

exportset	The set of variable *Quals* defined in the exporting process
stnm	The name of the state to which the implicit inputs are to be attached to.

Result See the definition of *Speclist*

Algorithm

Line 1	When through, return nothing (the function is recursive).
Line 4	Let *qual* denote a variable *Qual* in the set.
Line 5	Construct the AS_0 nextstate node which follows the implicit inputs.
Line 6	Decompose the descriptor of the variable.
Line 7	If it is not an exported variable then continue to consider the next variable.
Line 10	Extract the name of the exported variable.

Line 11	Decompose the *SignalnamesD* descriptor (see the definition of *Exportmap*) common for all export variables in the system having that name (*nm*) and that sort (*tq*). *imp* denotes the name of the implicit variable attached to the export variable, *xq* is the name of the xtQUERY signal and *xr* is the name of the xtREPLY signal.
Line 13	Create the AS_0 output node which sends the xtREPLY signal.
Line 14	Create the transition which directly follows the input node.
Line 15	Create the input node which contains the xtQUERY signal.
Line 16	Return the *Spec* containing the complete input transition (*xqinput*) joined with the *Speclist* for the rest of the export variables

transform-input(*stnm*, *statespecl*)(*dict*) $\triangleq$ (3.7.6.4)

```
 1  (if statespecl = ⟨⟩ then
 2     ({},dict)
 3    else
 4     (let (scope, node) = hd statespecl in
 5      let labeldict' =
 6          if is-ServiceD(dict(scope))
 7            then s-Labeldict(dict(scope))
 8            else dict(LABELDICT) in
 9      let dict' = dict + [SCOPEUNIT ↦ scope,
10                          LABELDICT ↦ labeldict'] in
11       cases node:
12        (mk-Priinput0(inputvars, trans),
13         mk-Inputspec0(inputvars, , trans)
14           → (let (as1 set', dict'') = transform-input-transition(stnm, inputvars, trans)(dict') in
15              let (as1 rest, dict''') = transform-input(stnm, tl statespecl)(dict'') in
16              (as1 set' ∪ as1 rest, dict''')))))
```

type: *Statename*$_0$ *Speclist* → *Dict* → *Input-node*$_1$-**set** *Dict*

Objective Transform the input transitions of a state and update *dict* with information about implicit variables and with output signals.

Parameters

stnm	The name of the state
statespecl	The *Speclist* for the state.

Result The set of AS_1 input nodes for the state.

Algorithm

Line 1	When through, return the empty set.
Line 4	Decompose the next *Spec* to be considered.
Line 5	Extract the *Labeldict* for the scopeunit. If the input transition originates from a service (before they were merged) then extract it from the service descriptor.
Line 9	Update SCOPEUNIT in *Dict* to denote the scopeunit from which the input transition originates and update LABELDICT to denote the *Labeldict* of the scopeunit.
Line 12-13	decompose the present input transition.
Line 14	Create a set of AS_1 input nodes for all signals in the AS_0 input node.
Line 15	Transform the rest of the input transitions
Line 16	Add the newly created input nodes to the rest.

transform-input-transition(*stnm*, *inpvars*, *trans*)(*dict*) $\triangleq$ (3.7.6.5)

```
1  (if inpvars = ⟨⟩ then
2     ({}, dict)
3   else
4    (let (vl1, sigq) = transform-inputvars(hd inpvars)(dict) in
5     let as1 sigid = make-as1-identifier(sigq)(dict) in
6     let (trans1, dict') = transform-transition(stnm, trans, ⟨⟩)(dict) in
7     let inputnode = mk-Input-node1(as1 sigid, vl1, trans1) in
8     let (inprest, dict'') = transform-input-transition(stnm, tl inpvars, trans)(dict') in
9     ({inputnode} ∪ inprest, dict'')))
```

type : *Statename*$_0$ *Inputvars*$_0{}^+$ *Transition*$_0$ $\rightarrow$ *Dict* $\rightarrow$ *Input-node*$_1$-**set** *Dict*

Objective Transform an input transition into AS_1

Parameters

stnm	The name of the state
inpvars	The list of pairs of signal and parameters for the input
trans	The transition which follows the input

Result The set of AS_1 input nodes. There are as many input nodes as there are pairs of signal and parameters in the AS_0 input nodes.

Algorithm

Line 1	When through all the pairs of signal and parameters then return the empty set.
Line 4	Construct the AS_1 parameter list (vl_1) and the signal *Qual* from the first pair of AS_0 signal and parameters.
Line 5	Construct the AS_1 identifier of the signal.
Line 6	Transform the transition.
Line 7	Construct the AS_1 input node for the signal.
Line 8	Construct the AS_1 input nodes for the rest of pairs of signal
Line 9	Return the AS_1 input nodes joined with the rest of AS_1 input nodes. and parameters

transform-inputvars(**mk**-*Inputvars*$_0$(*sigid*, *varl*))(*dict*) $\triangleq$ (3.7.6.6)

```
1  (let squal = signal-qual(sigid)(dict) in
2   let tquallist = cases dict(squal):
3                     (mk-SignalD(tquallist, , ) → tquallist,
4                      mk-TimerD(tquallist, ) → tquallist) in
5   if (len varl = len tquallist) then
6     (let vl = transform-inputvarlist(varl, tquallist)(dict) in
7      (vl, squal))
8    else
9     exit("§2.6.4: Wrong number of parameters in input node"))
```

type : *Inputvars*$_0$ $\rightarrow$ *Dict* $\rightarrow$ [*Variable-identifier*$_1$]* *Signalqual*

Objective Construct the AS_1 variable list of an input node and extract the *Qual* of the input signal

Parameters

sigid	The AS_0 signal identifier
varl	The AS_0 variable list

Algorithm

Line 1	Let *squal* denote the *Qual* of the signal.
Line 2-4	Let *tquallist* denote the list of sorts for the parameters for the signal.
Line 5	The length of the sort list must be equal to the length of the variable list.
Line 6-7	Transform the variable list into AS_1 and return it together with the signal *Qual*

transform-inputvarlist(*varl*, *sortlist*)(*dict*) $\triangleq$ **(3.7.6.7)**

```
1  (if varl = ⟨⟩ then
2     ⟨⟩
3   else
4    (let restvl = transform-inputvarlist(tl varl, tl sortlist)(dict) in
5     if hd varl = nil then
6       ⟨nil⟩ ⌢ restvl
7      else
8       (let vqual = get-visible-variable(hd varl, {get-parent(hd sortlist)(dict)}, false)(dict) in
9        let mk-VarD(,,,,nqual) = dict(vqual) in
10       let as1id = make-as1-identifier(nqual)(dict) in
11       ⟨as1id⟩ ⌢ restvl)))
```

type: $[Varid_0]^*\ Sortqual^* \rightarrow Dict \rightarrow [Variable\text{-}identifier_1]^*$

Objective Construct the AS_1 variable list to be attached to an input node

Parameters

varl	The AS_0 variable list
sortlist	The sort list for the signal

Algorithm

Line 1	When through, return the empty list.
Line 4	Transform the tail of the list.
Line 5	If the first variable is omitted then return a **nil** object.
Line 8	Extract the *Qual* of the first identifier by looking at the context
Line 10	Construct the AS_1 variable identifier
Line 11	Return the AS_1 variable identifier joined with the variable identifiers for the rest of the list

insert-importact(*impact*, $signalset_1$) $\triangleq$ **(3.7.6.8)**

```
1  (let mk-Input-node1(replyid, val, trans) ∈ signalset1 in
2   let mk-Transition1(nodes, term) = trans in
3   if is-Decision-node1(impact) then
4     {mk-Input-node1(replyid, val, mk-Transition1(nodes, impact))}
5    else
6     {mk-Input-node1(replyid, val, mk-Transition1(nodes ⌢ ⟨impact⟩, term))})
```

type: $(Graph\text{-}node_1 \mid Decision\text{-}node_1)\ Input\text{-}node_1\text{-}\mathbf{set} \rightarrow Input\text{-}node_1\text{-}\mathbf{set}$

Objective Insert an AS_1 action in an AS_1 input node containing the xtREPLY signal. This action was constructed in connection with the transformation of an expression which contain import expressions. As import expressions are expanded into AS_0 states, the action could not be inserted until the state has been transformed

Parameters

impact	The AS_1 action to be inserted
signalset$_1$	The AS_1 input node set which consist of the xtREPLY input node only

Result The modified AS_1 input node set containing the xtREPLY input node

Algorithm

Line 1	Decompose the AS_1 input node.
Line 2	Decompose the contained transition.
Line 3	If the node to be inserted is an decision (in the case where a question contained import expression) then
Line 4	Replace the terminator by the decision action else
Line 6	Append the action to the action list

3.7.7 Transformation of Transition Strings

transform-transition(*tnm*, mk-*Transition*$_0$(*actl*, *term*), *nodes*$_1$)(*dict*) $\triangleq$ (3.7.7.1)

```
1  (if actl = ⟨⟩ then
2     (let mk-Termstmt0(, t) = term in
3       cases t:
4       (mk-Nextstate0(stnm)
5          → if tnm = nil ∧ stnm = nil
6               then exit("§4.9: Dash nextstate in initial transition")
7               else (let statedict = dict(STATEDICT) in
8                     let stnm' = if stnm = nil then tnm else stnm in
9                     let nextstate =
10                       if tnm = nil ∧ is-ServiceD(dict(dict(SCOPEUNIT))) ∧ stnm' ∉ dom statedict then
11                         (let (,servicelist) = dict(SERVICES) in
12                          extract-initial-state(servicelist, stnm', ⟨⟩)(dict))
13                        else
14                         if is-ServiceD(dict(dict(SCOPEUNIT))) ∧ stnm' ∉ dom statedict
15                           then extract-servicestate(tnm, stnm')(dict)
16                           else stnm' in
17                    if nextstate ∈ dom statedict then
18                      if is-ContenablestateD(statedict(nextstate)) then
19                        (let trans = s-Transition0(statedict(nextstate)) in
20                         transform-transition(tnm, trans, nodes1)(dict))
21                       else
22                        (mk-Transition1(nodes1, mk-Nextstate-node1(name-to-name1(nextstate))), dict)
23                     else
24                      exit("§2.6.7.2.1: Name in nextstate must denote a defined state")),
25       mk-Stop0()
26         → (mk-Transition1(nodes1, mk-Stop-node1()), dict),
27       mk-Return0()
28         → (mk-Transition1(nodes1, mk-Return-node1()), dict),
29       mk-Join0(l)
30         → (let labeldict = dict(LABELDICT) in
31            if l ∈ dom labeldict then
32              (let trans = labeldict(l) in
33               transform-transition(tnm, trans, nodes1)(dict))
34            else
35             exit("§2.6.7.2.2: Label in join is not defined"))))
36   else
37    (let (tran1, dict') = transform-act(tnm, actl, term, nodes1)(dict) in
38      cases tran1:
39      (mk-Transition1(, )
40         → (tran1, dict'),
41       T → transform-transition(tnm, mk-Transition0(tl actl, term), nodes1 ⌢ tran1)(dict'))))
```

type: [*Statename*$_0$] *Transition*$_0$ *Graph-node*$_1$* → *Dict* → *Transition*$_1$ *Dict*

Objective Transform a transition into AS$_1$

Parameters

tnm	The name of the state containing the transition. If *tnm* is **nil** then the transition is an initial transition
actl	The action list of the transition
term	The terminator of the transition
nodes$_1$	The nodes (from the transition) which already have been transformed

Result The AS$_1$ transition

Algorithm

Line 1-35	When through the action list then transform the terminator.
Line 4	If the terminator is a nextstate then it must not contain a dash if it is contained in the initial transition.
Line 7	Let *statedict* denote the *Statedict* of the process.
Line 8	Let *stnm'* denote the name in the nextstate if it is specified, otherwise it denotes the name of the state leading to the nextstate.
Line 9-16	Let *nextstate* denote the name of the next state.
Line 9-12	If it is the initial transition and the scopeunit is a service and it is not a nextstate leading to an import state then extract the state name of the next state by considering the nextstate nodes in the initial terminator of all services.
Line 14	If the scopeunit is a service and the state name cannot be found in *Statedict* (because the explicit states in the services have been merged and each new state in statedict have a distinct new name) then extract the distinct new state by considering the old service state name and the new service state name.
Line 17	The name of the next state must be defined.
Line 18-20	If the nextstate leads to a state which has been replaced by a number of enabling condition states then the *Statedict* contains a transition only and this transition replaces the nextstate node.
Line 22	Construct the AS_1 transition containing the transformed actions ($nodes_1$) and the nextstate node.
Line 25	If the terminator is a stop then construct the AS_1 transition containing a stop node.
Line 27	If the terminator is a return then construct the AS_1 transition containing a return node.
Line 29	If the terminator is a join then let *labeldict* denote the *Labeldict* for the scopeunit.
Line 31	The label must be defined.
Line 32	Extract the transition where the label is defined.
Line 33	Replace the join by that transition.
Line 37	If there are more actions in the transition then transform the first action in the action list. Note that although only the first action is transformed, the full action list is given as parameter.
Line 39	If the result of transforming the first action is a full transition then the first action contains import expressions and the transition is returned.
Line 41	Otherwise transform the rest of the actions

extract-initial-state(*servicelist*, *nextstate*, *statelist*)(*dict*) $\triangleq$ (3.7.7.2)

```
 1  (if servicelist = ⟨⟩ then
 2     (let (statetuplemap, ) = dict(SERVICES) in
 3      if statelist ∈ dom statetuplemap then
 4        statetuplemap(statelist)
 5       else
 6        exit("§2.6.7.2.1: Name in nextstate must denote a defined state"))
 7    else
 8     if hd servicelist = dict(SCOPEUNIT) then
 9       extract-initial-state(tl servicelist, ..., statelist ⌢ ⟨nextstate⟩)(dict)
10      else
11       (let mk-ServiceD(trans, , , , ) = dict(hd servicelist) in
12        let mk-Transition0( , mk-Termstmt0( , mk-Nextstate0(stnm))) = trans in
13        extract-initial-state(tl servicelist, nextstate, statelist ⌢ ⟨stnm⟩)(dict)))
```

type : *Servicetuple Statename*$_0$ *Statename*$_0$* $\rightarrow$ *Dict* $\rightarrow$ *Statename*$_0$

Objective Extract the first (composite) state in the case where the states consist of merged service states and one of the services contains an initial transition

Parameters

servicelist	The list of service *Quals* matching the state name tuple (*statelist*)
nextstate	The name of the next service state
statelist	The state name tuple

Result The deduced (unique) state name

Algorithm

Line 1	When having considered the initial nextstates of all services then
Line 2	Let *statetuplemap* denote the relation between the state name tuples and the associated unique names.
Line 3	The constructed state name tuple for nextstates must be included in *statetuplemap*, otherwise there is one of the nextstates which contains an undefined state name.
Line 4	Return the unique state name corresponding to the service nextstates.
Line 8-9	If the service to be considered is the current one then add the nextstate name to the state name tuple before considering the rest of the services. The name in the nextstate of the current service in not used by the function any longer (therefore "...").
Line 11	Let trans denote the initial transition of the service to be considered (it has an empty action list).
Line 12	Let *stnm* denote the name of the state in its initial nextstate node.
Line 13	Add the state name to the state name tuple before considering the rest of the services

extract-servicestate(*oldstate*, *stnm*)(*dict*) $\triangleq$ (3.7.7.3)

```
1  (let (statetuplemap, servicelist) = dict(SERVICES) in
2   let statelist be s.t. statelist ∈ dom statetuplemap ∧ statetuplemap(statelist) = oldstate in
3   let index be s.t. servicelist[index] = dict(SCOPEUNIT) in
4   let newstatelist = ⟨statelist[i] | 1 ≤ i < index⟩ ⌢
5       ⟨stnm⟩ ⌢
6       ⟨statelist[i] | index < i ≤ len statelist⟩ in
7   if newstatelist ∈ dom statetuplemap then
8     statetuplemap(newstatelist)
9    else
10    exit("§2.6.7.2.1: Name in nextstate must denote a defined state"))
```

type: $Statename_0$ $Statename_0 \rightarrow Dict \rightarrow Statename_0$

Objective From the name of an old (but unique) state and the name of a new service state extract the name of the unique state which is defined in *Statedict*

Algorithm

Line 1	Let *statetuplemap* denote the relation between the state name tuples and the unique state names. Let *servicelist* denote the list of service *Quals* which matched the state name tuples.
Line 2	Let *statelist* denote the state name tuple which correspond to the old state name.
Line 3	Let *index* denote the index to the current service.
Line 4-6	Overstrike the state name tuple with the new state name at the position indicated by index.

Line 7-8 If there exist a unique state name for this state name tuple then return the unique name else the new service state name is not defined

3.7.8 Transformation of Action Statements

In this subsection, action statements are transformed into AS_1. To every transformation function the name of the state containing the action statement, the action statements following the action statement, the terminator following the rest of the action statements and the action statements which already have been transformed are given (see *transform-act*). The import expressions in the action statement are represented by an implicit variable in the AS_1 action statement. If the action statement contained any import expressions then a complete $Transition_1$ containing the previously transformed nodes and a nextstate to the implicit state is returned instead and *Statedict* is updated to include the *StateD* descriptor which contains information about the implicit state, the action statements following the action statement, the terminator following the action statements, the name of the state following the action statements and the transformed node (see the definitions of *StateD* and especially *Importstateinf*)

transform-act(*tnm*, *actl*, *term*, $nodes_1$)(*dict*) $\triangleq$ (3.7.8.1)

```
1  (let mk-Actstmt0(, act) = hd actl in
2   cases act:
3    (mk-Task0(stmtl)  → transform-stmtlist(tnm, stmtl, tl actl, term, nodes1)(dict),
4     mk-Output0(, , ) → transform-output(tnm, act, tl actl, term, nodes1)(dict),
5     mk-Prioutput0()  → transform-prioutput(tnm, act, tl actl, term, nodes1)(dict),
6     mk-Create0(, )   → transform-create(tnm, act, tl actl, term, nodes1)(dict),
7     mk-Decision0(, , ) → transform-decision(tnm, act, nodes1)(dict),
8     mk-Option0(, , ) → transform-option(tnm, act, nodes1)(dict),
9     mk-Reset0()      → transform-reset(tnm, act, tl actl, term, nodes1)(dict),
10    mk-Set0()        → transform-set(tnm, act, tl actl, term, nodes1)(dict),
11    mk-Export0(idl)  → (⟨transform-export(idl[i])(dict) | 1 ≤ i ≤ len idl⟩, dict),
12    mk-Call0(, )     → transform-call(tnm, act, tl actl, term, nodes1)(dict)))
```

type : [*Statename*$_0$] *Actstmt*$_0^+$ [*Termstmt*$_0$] *Graph-node*$_1^*$ → *Dict* → (*Transition*$_1$ | *Graph-node*$_1^+$) *Dict*

Objective Transform an action statement into AS_1

Parameters

tnm	The name of the state to which the action statement belongs. The name is used for replacing nextstate dash
actl	The action statement list. The first action statement in this list is the one which is transformed. The rest is only used if the action statement contains import expressions
term	The terminator which follows the action statement list. The terminator is only used if the action statement contains import expressions
$nodes_1$	The previous nodes which already have been transformed.

Result A *Graph-node*$_1$ if the action statement contains no import expressions, otherwise a *Transition*$_1$ which terminates at the implicit import state. *Dict* is updated to include information about the output signals and the implicit variables

Algorithm

Line 1-2 Transform an action (*act*) which is either.

Line 3 A task.

Line 4	Or an output.
Line 5	Or a priority output.
Line 6	Or a create request.
Line 7	Or a decision.
Line 8	Or an option.
Line 9	Or a timer reset.
Line 10	Or a timer set.
Line 11	Or an export.
Line 12	Or a procedure call
	Note that the label in the $Actstmt_0$ is not used any longer as the association information about it is contains in the *Labeldict*

transform-stmtlist(*tnm*, *stmtl*, *actl*, *term*, $nodes_1$)(*dict*) $\triangleq$ **(3.7.8.2)**

```
1  (if stmtl = ⟨⟩ then
2     (nodes1, dict)
3   else
4     (let (as1 task, d) = cases hd stmtl:
5                            (mk-Assignstmt0(,)
6                               → transform-assign(hd stmtl)(dict),
7                             mk-Text0(text)
8                               → (transform-informal(text), dict)) in
9      if d(IMPORTLIST) = ⟨⟩ then
10       transform-stmtlist(tnm, tl stmtl, actl, term, nodes1 ⌢ as1 task)(d)
11     else
12       (let rest = if tl stmtl = ⟨⟩ then ⟨⟩ else ⟨mk-Task0(tl stmtl)⟩ in
13        let (outputtrans, d') = transform-import(tnm, as1 task, rest ⌢ actl, term)(d) in
14        transform-transition(tnm, outputtrans, nodes1)(d'))))
```

type : [$Statename_0$] ($Assignstmt_0^*$ | $Text_0^*$) $Actstmt_0^*$ [$Termstmt_0$] $Graph\text{-}node_1^*$ → *Dict* → (*Transition*$_1$ | $Graph\text{-}node_1^+$) *Dict*

Objective Transform a statement list into AS_1

Parameters

stmtl	The statement list to be transformed
tnm, *actl*, *term*, $nodes_1$	See *transform-act* and the introduction to this subsection

Result See *transform-act* and the introduction to this subsection

Algorithm

Line 1	When through, return the list of AS_1 task nodes each containing a statement and return the updated *Dict*.
Line 4-7	Let $as_1\,task$ denote the task of the first statement in the list.
Line 5	If the first statement is an assignment then transform it.
Line 7	If the first statement is informal text then transform it and leave *Dict* unchanged.
Line 9-10	If there are no import expressions in the statement then transform the rest of the statements else
Line 12	Construct a task which contains the rest of the statements (if there are any).
Line 13	Update *Statedict* to include information about the implicit state and construct the transition which contains the action statements which previously have been transformed, the output of the xt-QUERY signal and the nextstate to the implicit state.

Line 14 Transform the constructed transition

transform-informal(*text*) $\triangleq$ (3.7.8.3)

```
1  (let mk-Text0(t) = text in
2   mk-Task-node1(mk-Informal-text1(t)))
```

type: $Text_0 \rightarrow Informal\text{-}text_1$

Objective Transform informal text into its AS_1 representation.

transform-assign(mk-*Assignstmt*$_0$(*vid*, *expr*))(*dict*) $\triangleq$ (3.7.8.4)

```
1  (if is-Id0(vid) then
2     (let sortset = all-visible-sorts(dict) in
3      let (,sortset',) = transform-expr(expr, EXPRESSION, sortset)(dict) in
4      let qual = get-visible-variable(vid, sortset', false)(dict) in
5      let mk-VarD(q,,,, vqual) = dict(qual) in
6      let (as1tree,, d) = transform-expr(expr, EXPRESSION, {get-parent(q)(dict)})(dict) in
7      let as1id = make-as1-identifier(vqual)(dict) in
8      let as1assign = mk-Assignment-statement1(as1id, as1tree) in
9      (mk-Task-node1(as1assign), d))
10    else
11     transform-modifyassign(vid, expr)(dict))
```

type: $Assignstmt_0 \rightarrow Dict \rightarrow Task\text{-}node_1\ Dict$

Objective Transform an assignment statement into AS_1

Parameters The assignment statement containing

vid	The variable to be assigned
expr	The right-hand side expression

Result The AS_1 assignment and a *Dict* updated to include information about import expressions in *expr* and implicit variables

Algorithm

Line 1	If the variable is an identifier then
Line 2-4	Construct its *Qual* by extracting the sorts which matches the expression (line 3) and by using those sorts when resolving the right-hand side by context (line 4).
Line 5	Decompose the variable descriptor. let *q* denote the *qual* of its sort and let *vqual* denote its unique *Qual* (differs in case of services).
Line 6	Transform the right-hand side expression which must be of the sort *q*.
Line 7	Construct the AS_1 identifier from its unique *Qual*.
Line 8	Construct the AS_1 assignment statement.
Line 9	Construct and return the task which contains the assignment statement.
Line 11	If the variable is not an identifier then the assignment is a shorthand notation for the MODIFY! or field modify operator

transform-modifyassign(*vid*, *expr*)(*dict*) $\triangleq$ (3.7.8.5)

```
1  (let makeexpr(var) =
2      if is-Indexedvar0(var) then
3        mk-Operatorapp0(makeexpr(s-Variable0(var)), s-Exprlist0(var))
4       else
5        if is-Fieldvar0(var) then
6          mk-Selectexpr0(makeexpr(s-Expr0(var)), s-Name0(var))
7         else
8          var in
9   cases vid:
10   (mk-Indexedvar0(vid', exprlist)
11      → (let opid = mk-Id0(⟨⟩, mk-Name0("MODIFY", EXCLAMATION)) in
12         let vid'' = makeexpr(vid') in
13         let righthandside = mk-Operatorapp0(opid, ⟨vid''⟩ ⌢ exprlist ⌢ ⟨expr⟩) in
14         let (as1 node, dict') =
15             (trap exit with (nil, dict) in
16              transform-assign(mk-Assignstmt0(vid', righthandside))(dict)) in
17         let (as1 node', dict'') =
18             if is-Id0(exprlist[len exprlist]) ∧
19              s-Qualifier0(exprlist[len exprlist]) = ⟨⟩ then
20              (let field = s-Name0(exprlist[len exprlist]) in
21               let vid'' = if len exprlist = 1 then
22                                vid'
23                           else
24                                mk-Indexedvar0(vid', ⟨exprlist[i] | 1 ≤ i ≤ len exprlist − 1⟩) in
25               let vid''' = mk-Fieldvar0(vid'', field) in
26               trap exit with (nil, dict) in
27               transform-assign(mk-Assignstmt0(vid''', expr))(dict))
28              else
29               (nil, dict) in
30         (as1 node = nil ∧ as1 node' ≠ nil
31            → (as1 node', dict''),
32          as1 node ≠ nil ∧ as1 node' = nil
33            → (as1 node, dict'),
34          T → exit("§5.5.3: None or multiple expansions of modify assignment"))),
35    mk-Fieldvar0(vid', mk-Name0(str, ))
36      → (let opid = mk-Id0(⟨⟩, mk-Name0(str ⌢ "MODIFY", EXCLAMATION)) in
37         let righthandside = mk-Operatorapp0(opid, ⟨vid', expr⟩) in
38         transform-assign(mk-Assignstmt0(vid', righthandside))(dict))))
```

type: $Variable_0\ Expr_0 \rightarrow Dict \rightarrow Task\text{-}node_1\ Dict$

Objective Transform an assignment of an indexed variable or a field variable into AS_1

Parameters

vid	The left-hand side variable
expr	The right-hand side expression

Result The task node which contains the AS_1 representation of the assignment shorthand

Algorithm

Line 1-8	Construct a recursive function which converts an indexed left-hand side to an indexed right-hand side (i.e. an operator application) and a field left-hand side to a field right-hand side (a $Selectexpr_0$)
Line 10-16	Expand the shorthand of an indexed variable assignment. For example

```
v(e1)(e2,e3) := e4
```

is expanded into

```
v(e1) := MODIFY!(v(e1),e2,e3,e4)
```

which via *transform-assign* is expanded into

```
v := MODIFY!(v,e1,MODIFY!(v(e1),e2,e3,e4))
```

before it is transformed into AS_1. If the transformation fails, the error is trapped because it may mean that the assignment is the other shorthand (constructed line 17-29).

Line 11 Construct the identifier of the MODIFY! operator and

Line 12 Let *vid''* denote the converted left-hand side.

Line 13 Construct the new right-hand side.

Line 16 Transform the new assignment statement.

Line 17-29 Expand the parenthesis form of a field select assignment. For example

```
v(e1)(e2,e3) := e4
```

is expanded into

```
v(e1)(e2)!e3 := e4
```

provided e3 is an unqualified identifier. This constructed $Fieldvar_0$ is transformed into AS_1 and if the transformation fails, the error is trapped because it may mean that the assignment is the other shorthand (constructed line 11-16).

Line 17-19 Unless the last expression in the expression list is an unqualified identifier then this shorthand is not possible.

Line 20 Extract the name of the field

Line 21-24 If there are more expressions in the expression list then construct another $Indexedvar_0$ else return the $Variable_0$ (*vid'*).

Line 25 Construct the field select construct.

Line 26-27 Try to transform the field select assignment.

Line 30-34 There must be one and only one of the two possible expansions which is well-formed.

Line 35-38 Expand a field select assignment for example

```
a!b!c := e
```

is expanded into

```
a!b := cMODIFY!(a!b,e)
```

which via *transform-assign* is expanded into

```
a := bMODIFY!(a,cMODIFY!(a!b,e))
```

before it is transformed into AS_1.

Line 36 Construct the identifier of the modify operator, which modifies the field with the spelling *str*.

Line 37 Construct the new right-hand side.

Line 38 Transform the new assignment statement

transform-reset(*tnm*, *resetnode*, *actl*, *term*, *nodes*$_1$)(*dict*) $\triangleq$ **(3.7.8.6)**

```
1  (let mk-Reset0(resetlist) = resetnode in
2   let (actl1, d) = transform-reset-elems(tnm, resetlist, actl, term, nodes1)(dict) in
3   if tl resetlist = ⟨⟩ ∨ is-Transition1(actl1) then
4     (actl1, d)
5    else
6     transform-reset(tnm, mk-Reset0(tl resetlist), actl, term, nodes1 ⌢ actl1)(d))
```

type: [*Statename*$_0$] *Reset*$_0$ *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* →
Dict → (*Transition*$_1$ | *Graph-node*$_1$*) *Dict*

Objective Transform a reset action into AS$_1$

Parameters

tnm,	
actl,	
term,	
nodes$_1$	See *transform-act*
resetnode	The reset action to be transformed

Result See *transform-act*

Algorithm

Line 1	Let *resetlist* denote the list of timers to be reset.
Line 2	Transform the first timer to be reset.
Line 3	If this timer reset is the last in the list (the function is recursive) or if a transition is returned (because of import export expressions in the expression list) then stop the transformation (if the expression list contains import expressions then the rest of the timers will be transformed when the implicit import state is transformed).
Line 6	Else transform the rest of the timer resets

transform-reset-elems(*tnm*, *resetlist*, *actl*, *term*, *nodes*$_1$)(*dict*) $\triangleq$ **(3.7.8.7)**

```
1  (let mk-Resetelem0(timerid, actparm) = hd resetlist in
2   let squal = get-visible-qual(timerid, SIGNAL)(dict) in
3   if is-SignalD(dict(squal)) then
4     exit("§2.8: Identifier in reset action is not a timer")
5    else
6     (let mk-TimerD(tplist, ) = dict(squal) in
7      if len actparm = len tplist then
8        (let (as1 actparm, d) = transform-actparms(tplist, actparm, EXPRESSION)(dict) in
9         let as1 id = make-as1-identifier(squal)(dict) in
10        let as1 tree = mk-Reset-node1(as1 id, as1 actparm) in
11        if d(IMPORTLIST) = ⟨⟩ then
12          (⟨as1 tree⟩, d)
13         else
14          (let rest = if tl resetlist = ⟨⟩ then ⟨⟩ else ⟨mk-Reset0(tl resetlist)⟩ in
15           let (outputtrans, d') = transform-import(tnm, as1 tree, rest ⌢ actl, term)(d) in
16           transform-transition(tnm, outputtrans, nodes1)(d')))
17       else
18        exit("§2.8: Parameter list length in reset action must equal the length specified in the definition")))
```

type: [*Statename*$_0$] *Resetelem*$_0$* *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* →
Dict → (*Transition*$_1$ | *Graph-node*$_1$*) *Dict*

Objective Transform a timer reset into AS$_1$

Parameters See *transform-reset*

Result See *transform-reset*

Algorithm

Line 1	Decompose the $Resetelem_0$ which is going to be transformed.
Line 2	Extract the *Qual* of the identifier specified.
Line 3	The identifier must denote a timer signal.
Line 6	Let *tplist* denote the sorts of the expression list specified for the timer.
Line 7	There must be as many expressions in the list as there are sorts for the timer.
Line 8	Transform the expressions.
Line 9	Construct the AS_1 identifier of the timer.
Line 10	Construct the AS_1 reset node.
Line 11	If there are no import expressions in the expression list then
Line 12	Return the AS_1 reset node else
Line 14	Construct an AS_0 reset action of the rest of the timer resets. This action will be attached to the implicit import state.
Line 15	Update *Dict* with the import state and construct the AS_0 transition which leads to the import state.
Line 16	Transform the transition

$transform\text{-}set(tnm, setnode, actl, term, nodes_1)(dict) \triangleq$ **(3.7.8.8)**

```
1  (let mk-Set0(setlist) = setnode in
2   let (actl1, d) = transform-set-elems(tnm, setlist, actl, term, nodes1)(dict) in
3   if tl setlist = ⟨⟩ ∨ is-Transition1(actl1) then
4     (actl1, d)
5    else
6     transform-set(tnm, mk-Set0(tl setlist), actl, term, nodes1 ⌢ actl1)(d))
```

type : $[Statename_0]\ Set_0\ Actstmt_0^*\ [Termstmt_0]\ Graph\text{-}node_1^* \rightarrow Dict \rightarrow (Transition_1 \mid Graph\text{-}node_1^*)\ Dict$

Objective Transform a set action into AS_1

Parameters

tnm,	
actl,	
term,	
$nodes_1$	See *transform-act*
resetnode	The set action to be transformed

Result See *transform-act*

Algorithm

Line 1	Let *setlist* denote the list of timers to be set.
Line 2	Transform the first timer to be set.
Line 3	If this timer set is the last in the list (the function is recursive) or if a transition is returned (because of import export expressions in the expression list) then stop the transformation (if the expression list contains import expressions then the rest of the timers will be transformed when the implicit import state is transformed).
Line 6	Else transform the rest of the timer sets

transform-set-elems(*tnm*, *setlist*, *actl*, *term*, *nodes*$_1$)(*dict*) $\triangleq$ (3.7.8.9)

```
 1  (let mk-Setelem0(timeexpr, timerid, actparm) = hd setlist in
 2   let timesort = get-predef-sort("TIME")(dict) in
 3   let (as1 expr, d) = transform-expr(timeexpr, EXPRESSION, {timesort})(dict) in
 4   let squal = get-visible-qual(timerid, SIGNAL)(dict) in
 5   if is-SignalD(dict(squal)) then
 6     exit("§2.8: Identifier in set action is not a timer")
 7    else
 8     (let mk-TimerD(tplist, ) = dict(squal) in
 9      if len actparm = len tplist then
10        (let (as1 actparm, d') = transform-actparms(tplist, actparm, EXPRESSION)(d) in
11         let as1 id = make-as1-identifier(squal)(dict) in
12         let as1 tree = mk-Set-node1(as1 expr, as1 id, as1 actparm) in
13         if d'(IMPORTLIST) = ⟨⟩ then
14           (⟨as1 tree⟩, d')
15          else
16           (let rest = if tl setlist = ⟨⟩ then ⟨⟩ else ⟨mk-Set0(tl setlist)⟩ in
17            let (outputtrans, d'') =
18                transform-import(tnm, as1 tree, rest ↷ actl, term)(d') in
19            transform-transition(tnm, outputtrans, nodes1)(d'')))
20       else
21        exit("§2.8: Parameter list length in set action must equals the length specified in the definition")))
```

type: [*Statename*$_0$] *Setelem*$_0$* *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* $\rightarrow$
Dict $\rightarrow$ (*Transition*$_1$ | *Graph-node*$_1$*) *Dict*

Objective Transform a timer set into AS$_1$

Parameters See *transform-set*

Result See *transform-set*

Algorithm

Line 1 Decompose the *Setelem*$_0$ which is going to be transformed.

Line 2 Let *timesort* denote the *Qual* of the time sort.

Line 3 Transform the time expression specified.

Line 4 Extract the *Qual* of the identifier specified.

Line 5 The identifier must denote a timer signal.

Line 8 Let *tplist* denote the sorts of the expression list specified for the timer.

Line 9 There must be as many expressions in the list as there are sorts for the timer.

Line 10 Transform the expressions.

Line 11 Construct the AS$_1$ identifier of the timer.

Line 12 Construct the AS$_1$ set node.

Line 13 If there are no import expressions in the expression list or in the time expression then

Line 14 Return the AS$_1$ set node else

Line 16 Construct an AS$_0$ set action of the rest of the timer sets. This action will be attached to the implicit import state.

Line 17 Update *Dict* with the import state and construct the AS$_0$ transition which leads to the import state.

Line 19 Transform the transition

transform-export(*id*)(*dict*) $\triangleq$ (3.7.8.10)

```
1  (let sortset = all-visible-sorts(dict) in
2   let qual = get-visible-variable(id, sortset, true)(dict) in
3   let mk-SignalnamesD(impnm, , ) = exportmap((s-Sortqual(dict(qual)), s-Name0(id))) in
4   let as1 id = make-as1-identifier(qual)(dict),
5       as1 id' = make-as1-identifier(impnm)(dict) in
6   ⟨mk-Task-node1(mk-Assignment-statement1(as1 id', as1 id))⟩)
```

type : $Varid_0 \rightarrow Dict \rightarrow Task\text{-}node_1$

Objective Transform an export action into an AS_1 assignment statement

Parameters

id	The AS_0 identifier of the exported variable

Result An AS_1 task containing the constructed assignment statement

Algorithm

Line 1-2	Construct the *Qual* of the exported variable by using the context.
Line 3	Let *impnm* denote the name of the implicit variable attached to the exported variable
Line 4-5	Construct the AS_1 identifiers of the explicit export variable ($as_1\,id$) and of the implicit export variable ($as_1\,id'$) and
Line 6	Return the task which assigns the explicit variable to the implicit variable

3.7.8.1 Transformation of Output Nodes

transform-prioutput(*tnm*, mk-$Prioutput_0$(*outputsigs*), *actl*, *term*, $nodes_1$)(*dict*) $\triangleq$ (3.7.8.1.1)

```
1  (let self = mk-Selfexpr0() in
2   let (actl1, d) = transform-output-elems(tnm, outputsigs, self, ⟨⟩, actl, term, nodes1, true)(dict) in
3   if tl outputsigs = ⟨⟩ ∨ is-Transition1(actl1) then
4     (actl1, d)
5    else
6     transform-prioutput(tnm, mk-Prioutput0(tl outputsigs), actl, term, nodes1 ⌢ actl1)(d))
```

type : $[Statename_0]\ Prioutput_0\ Actstmt_0^*\ [Termstmt_0]\ Graph\text{-}node_1^* \rightarrow$
$Dict \rightarrow (Transition_1 \mid Graph\text{-}node_1^*)\ Dict$

Objective Transform a priority output action into a sequence of AS_1 output actions (one for each signal mentioned)

Parameters

tnm,	
actl,	
term,	
$nodes_1$	See *transform-act*
outputsigs	A list of outputs contained in the priority output action

Result See *transform-act*

Algorithm

Line 1-6	The algorithm follows the same scheme as for *transform-reset* The argument **true** to the function *transform-output-elems* indicates that the signals are high priority signals

transform-output(*tnm*, *outnode*, *actl*, *term*, *nodes*$_1$)(*dict*) $\triangleq$ (3.7.8.1.2)

```
1  (let mk-Output0(outputsigs, expr, via) = outnode in
2   let (actl1, d) = transform-output-elems(tnm, outputsigs, expr, via, actl, term, nodes1, false)(dict) in
3   if tl outputsigs = ⟨⟩ ∨ is-Transition1(actl1) then
4     (actl1, d)
5    else
6     transform-output(tnm, mk-Output0(tl outputsigs, expr, via), actl, term, nodes1 ⌢ actl1)(d))
```

type : [*Statename*$_0$] *Output*$_0$ *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* $\rightarrow$
Dict $\rightarrow$ (*Transition*$_1$ | *Graph-node*$_1$*) *Dict*

Objective Transform a output action into a sequence of AS$_1$ output actions (one for each signal mentioned)

Parameters

tnm,	
actl,	
term,	
nodes$_1$	See *transform-act*
outputsigs	A list of outputs contained in the output action

Result See *transform-act*

Algorithm

Line 1-3	The algorithm follows the same scheme as for *transform-reset* The argument **false** to the function *transform-output-elems* indicates that the signals are not high priority signals

transform-output-elems(*tnm*, *sigs*, *expr*, *via*, *actl*, *term*, $nodes_1$, *isprioutput*)(*dict*) $\triangleq$ **(3.7.8.1.3)**

```
1  (let mk-Outputsig0(sigid, actparm) = hd sigs in
2   let instsort = get-predef-sort("PID")(dict) in
3   let (as1 expr, d) =
4       if expr = nil
5         then (nil, dict)
6         else transform-expr(expr, EXPRESSION, {instsort})(dict) in
7   let squal = signal-qual(sigid)(dict) in
8   let qual = process-or-service-level(dict(SCOPEUNIT)) in
9   let existreceiver = (∃q ∈ dom dict)(is-ServiceD(dict(q)) ∧
10                                      get-sur(q) = get-sur(qual) ∧
11                                      squal ∈ s-Priinputset(dict(q))) in
12  if (is-ProcessD(dict(qual)) ∧ isprioutput) ∨
13    (isprioutput ≠ existreceiver) then
14     exit("§4.10: Illegal use of high priority signal")
15    else
16     (let qualset = transform-via(squal, via)(dict) in
17      let as1 pathidset = make-as1 idset(qualset)(dict) in
18      let dict' = d + [OUTSIGNALS ↦ dict(OUTSIGNALS) ∪ {squal}] in
19      if is-TimerD(dict(squal)) then
20        exit("§2.7.4: Identifier in output action must denote a signal")
21       else
22        (let mk-SignalD(tplist, , ) = dict(squal) in
23         if len actparm = len tplist then
24           (let (as1 actparm, d') =
25                transform-actparms(tplist, actparm, EXPRESSION)(dict') in
26            let as1 id = make-as1-identifier(squal)(dict) in
27            let as1 tree = mk-Output-node1(as1 id, as1 actparm, as1 expr, as1 pathidset) in
28            if d'(IMPORTLIST) = ⟨⟩ then
29              (⟨as1 tree⟩, d')
30             else
31              (let rest = if tl sigs = ⟨⟩ then
32                            ⟨⟩
33                          else
34                            ⟨mk-Output0(tl sigs, expr, via)⟩ in
35               let (outputtrans, d'') =
36                   transform-import(tnm, as1 tree, rest ⌢ actl, term)(d') in
37               transform-transition(tnm, outputtrans, nodes1)(d'')))
38          else
39           exit("§2.7.4: Not the right number of arguments to output action"))))
```

type : [*Statename*$_0$] *Outputsig*$_0$* [*Expression*$_1$] *Via*$_0$ *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* *Bool* → *Dict* → (*Transition*$_1$ | *Graph-node*$_1$*) *Dict*

Objective Transform an output or a priority output into AS_1

Parameters

tnm, *actl*, *term*, $nodes_1$	See *transform-act*
sigs	The list of outputs which were contained in the output action. Only the first element in this list is transformed in this function
expr	The destination expression. If the output is a priority output then *expr* is SELF
via	The list of channel or signal route identifiers contained in the output node. If the output is a priority output then the list is empty

Result See *transform-act*

Algorithm

Line 1	Let *sigid* denote the signal identifier of the output to be transformed and let *actparm* denote the associated actual parameter list.
Line 2	Extract the *Qual* of the PiD sort.
Line 3-6	If the destination expression is specified then transform it into AS_1. It must be of the PiD sort (*instsort*).
Line 7	Let *squal* denote the *Qual* of the signal.
Line 8	Let *qual* denote the *Qual* of the surrounding service or process (not procedure)
Line 9-11	*existreceiver* is true if there exist a service (*q*) in the same process (line 10) which can receive the signal in an high priority input node.
Line 12-13	If the output is contained in a process then it must not be a high priority output (line 12) and if it is a high priority signal then there must exist a receiving service and versa versa.
Line 16-17	Construct the AS_1 VIA set.
Line 18	Update the OUTSIGNALS entry of *Dict* to include the output signal *squal*.
Line 19	The signal specified must not be a timer signal.
Line 22	Let *tplist* denote the sort list attached to the signal.
Line 23	The length of the actual parameter list must be equal to the length of the sort list.
Line 24	Transform the actual parameters.
Line 26	Construct the AS_1 identifier of the signal.
Line 27	Construct the AS_1 output node.
Line 28-29	If there are no import expressions in the destination expression and in the actual parameters then return the output node else
Line 31	Construct an AS_0 output action of the rest of the outputs. This action will be attached to the implicit import state.
Line 35	Update *Dict* with the import state and construct the AS_0 transition which leads to the import state.
Line 37	Transform the transition

transform-via(*signalqual*, *pathlist*)(*dict*) $\triangleq$ (3.7.8.1.4)

```
(let processlevel = process-level(dict(SCOPEUNIT)) in
 let servicelevel = process-or-service-level(dict(SCOPEUNIT)) in
 let mk-ProcessD(, sset, , connectmap) = dict(processlevel) in
 let explicitsigroutes = s-Explicitroutes(dict(get-sur(processlevel))) in
 if pathlist = ⟨⟩ then
   if ¬explicitsigroutes then
     {}
    else
     if (∃mk-SignalrouteD(endp1, endp2, sigset1, sigset2) ∈ dom dict)
        ((processlevel = endp1 ∧ signalqual ∈ sigset1) ∨
         (processlevel = endp2 ∧ signalqual ∈ sigset2)) then
       {}
      else
       if signalqual ∈ sset then
         {}
        else
         exit("§2.7.4: Signal in output has no possible receiver")
  else
   (let id = hd pathlist in
    let qual = (trap exit with get-visible-qual(id, CHANNEL)(dict) in
                get-visible-qual(id, SIGNALROUTE)(dict)) in
    let qualrest = if tl pathlist = ⟨⟩ then
                     {}
                   else
                    transform-via(signalqual, tl pathlist)(dict) in
    if qual ∈ qualrest then
      exit("§2.7.4: Signal route or channel specified twice in VIA set")
     else
      cases dict(qual):
       (mk-SignalrouteD(endp1, endp2, sigset1, sigset2)
          → (if (processlevel = endp1 ∧ signalqual ∈ sigset1) ∨
                (processlevel = endp2 ∧ signalqual ∈ sigset2) then
                if connectmap ≠ [] ∧ is-ServiceD(dict(servicelevel)) ∧
                  ¬(∃q ∉ connectmap(qual))
                    ((let mk-SignalrouteD(ep1, ep2, sigs1, sigs2) = dict(q) in
                      (servicelevel = ep1 ∧ signalqual ∈ sigs1) ∨
                      (servicelevel = ep2 ∧ signalqual ∈ sigs2))) then
                   exit("§4.10.2: No service signal routes are connected to signal route in VIA")
                  else
                   if is-ProcessD(dict(servicelevel)) ∧ connectmap ≠ []
                     then exit("§2.10.2: VIA illegal outside services when service signal routes defined")
                     else {qual} ∪ qualrest
              else
               exit("§2.7.4: Illegal VIA set")),
```

```
45      mk-ChannelD(,,,,)
46        → if explicitsigroutes then
47            exit("2.7.4: If signal routes are specified then channels cannot be mentioned in VIA")
48          else
49           (let blockqual = get-sur(processlevel) in
50            let mk-BlockD(,,, cmap) = dict(blockqual) in
51            if qual ∉ dom cmap
52              then exit("§2.7.4: Channel in VIA set is not connected to block")
53              else (let squalset =
54                      {q ∈ cmap(qual) | (let mk-SignalrouteD(endp1, endp2, sigset1, sigset2) =
55                                           dict(q) in
56                                         (processlevel = endp1 ∧ signalqual ∈ sigset1) ∨
57                                         (processlevel = endp2 ∧ signalqual ∈ sigset2))} in
58                    if squalset = {}
59                      then exit("§2.7.4: Signal cannot be conveyed by channel in VIA set")
60                      else squalset ∪ qualrest)))))
```

type: *Signalqual* Via_0 → *Dict* → *Qual*-**set**

Objective Transform the list of channel or signalroute identifiers specified in the VIA construct into a signal route *Qual* set

Parameters

signalqual	The *Qual* of the signal specified in the output
pathlist	The AS_0 via list

Result The signal route *Qual* set

Algorithm

Line 1	Let *processlevel* denote the *Qual* of the surrounding process (not procedure or service).
Line 4	Let *explicitsigroutes* equals **true** if signal routes are specified in the surrounding block.
Line 5	If no channel identifiers or signalroute identifiers are specified then
Line 6	if no signal routes are specified for the surrounding block then return the empty set else
Line 9-11	If there exist a signal route which includes the signal specified and which has the surrounding process as one of its endpoints then return the empty set indicating that the signal may follow any path else
Line 14	If explicit signal routes are specified and the signal is not contained in one of them then the signal must be part of the valid input signal set for the surrounding process.
Line 19-59	If signal routes or channels are specified in the VIA then
Line 19	Let *id* denote the identifier of the first channel or signal route in the list.
Line 20-21	Look up the identifier in *Dict* as a signal route. If it fails then it must denote a channel and look up the identifier in *Dict* as a channel.
Line 22	Transform the rest of the channels or signal routes in the via list.
Line 26	If the signal route or channel also occurs in the *Quals* of the rest of the via list then the channel or signal route has been specified twice.
Line 30-44	If the *Qual* denotes a signal route then the signal must be included in the signal route, the surrounding process must be one of its endpoints and if service signal routes are specified (i.e. *connectmap* non-empty) then there must exist a service signal route connected to the service and to the signal route (line 33-38).

Line 40	If service signal routes are specified then VIA cannot be used in procedures on process level.
Line 45	If the *Qual* denotes a channel then
Line 46-47	It must be because no signal routes are specified explicit
Line 49	Let *blockqual* denote the *Qual* of the surrounding block.
Line 50	Let *cmap* denote the *BlockconnectionD* for the block.
Line 51-52	The channel in the VIA must be connected to the block.
Line 53-57	Let *squalset* denote the signal route *Qual*s which are connected to the channel and for which the signal is outgoing.
Line 58-59	There must be at least one such signal route.
Line 60	Return the selected signal routes joined with the signal routes for the rest in the VIA set.

3.7.8.2 Transformation of Create Nodes

transform-create(*tnm*, mk-*Create*$_0$(*pid*, *exprlist*), *actl*, *term*, *nodes*$_1$)(*dict*) $\triangleq$ (3.7.8.2.1)

```
 1  (let pqual = get-visible-qual(pid, PROCESS)(dict) in
 2   let bqual = get-sur(pqual) in
 3   let as1id = make-as1-identifier(pqual)(dict) in
 4   let mk-ProcessD(tplist, , , ) = dict(pqual) in
 5   if get-sur(process-level(dict(SCOPEUNIT))) ≠ bqual then
 6     exit("§2.7.2: Created process must be defined in the same block")
 7    else
 8     (if len tplist ≠ len exprlist then
 9        exit("§2.7.2: Actual parameter length must be equal to formal parameter list length")
10       else
11        (let (actparmlist, d) = transform-actparms(tplist, exprlist, EXPRESSION)(dict) in
12         let as1tree = mk-Create-request-node1(as1id, actparmlist) in
13         if d(IMPORTLIST) = ⟨⟩ then
14           (⟨as1tree⟩, d)
15          else
16           (let (outputtrans, d') = transform-import(tnm, as1tree, actl, term)(d) in
17            transform-transition(tnm, outputtrans, nodes1)(d')))))
```

type : [*Statename*$_0$] *Create*$_0$ *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* $\rightarrow$
Dict $\rightarrow$ (*Transition*$_1$ | *Graph-node*$_1$$^+$) *Dict*

Objective Transform a create request action into AS_1

Parameters

tnm,	
actl,	
term,	
nodes$_1$	See *transform-act*
pid	The process identifier contained in the create request action
exprlist	The list of actual parameters

Result See *transform-act*

Algorithm

Line 1	Let *pqual* denote the *Qual* of the process identifier.
Line 2	Let *bqual* denote the *Qual* of the surrounding block.
Line 3	Construct the AS_1 identifier of the process identifier.
Line 4	Let *tplist* denote the sorts of the actual parameters.

Line 5	The surroundings of the process where the create request node is placed must be the same (block) as the surrounding of the process identifier.
Line 8	The length of the sort list must be equal to the length of the actual parameter list.
Line 11	Transform the actual parameters.
Line 12	Construct the AS_1 create request node.
Line 13-14	If the actual parameter list contains no import expressions then return the constructed AS_1 create request node.
Line 16	Else update *Dict* with the import state and construct the AS_0 transition which leads to the import state.
Line 17	Transform the transition

transform-actparms(*tplist*, *exprlist*, *context*)(*dict*) $\triangleq$ **(3.7.8.2.2)**

```
1  (if tplist = ⟨⟩ then
2     (⟨⟩,dict)
3   else
4     (let (as1 expr, , d) =
5          if hd exprlist = nil
6            then (nil, , dict)
7            else transform-expr(hd exprlist, context, {hd tplist})(dict) in
8      let (as1 rest, drest) = transform-actparms(tl tplist, tl exprlist, context)(d) in
9      (⟨as1 expr⟩ ⌢ as1 rest, drest)))
```

type: *ParameterD** *Actparmlist*$_0$ *Context* → *Dict* → [*Term*$_1$ | *Expression*$_1$]* *Dict*

Objective Transform the actual parameters of an output action or a create request action or an operator argument list into AS_1

Parameters

tplist	The sorts of the actual parameters
exprlist	The actual parameters
context	The context in which the expressions are transformed. In the case of output action or create request action, the context is always EXPRESSION.

Result A list of optional expressions in the case of output action or create request action, and a list of terms or expressions in the case of operator actual parameters. *Dict* possibly updated to include information about import expressions in the actual parameter list

Algorithm

Line 1-2	When through, return the empty list and the (unchanged) *Dict*.
Line 4-7	Transform the first actual parameter if it is present in the actual parameter list.
Line 8	Transform the rest of the actual parameters.
Line 9	Return the first actual parameter joined with the rest of the actual parameters

3.7.8.3 Transformation of Call Nodes

transform-call(*tnm*, mk-*Call*$_0$(*procid*, *actparmlist*), *actl*, *term*, *nodes*$_1$)(*dict*) $\triangleq$ (3.7.8.3.1)

```
 1  (let pqual = get-visible-qual(procid, PROCEDURE)(dict) in
 2   let mk-ProcedureD(formparmlist, nqual) = dict(pqual) in
 3   let as1 id = make-as1-identifier(nqual)(dict) in
 4   if len actparmlist = len formparmlist then
 5     (let (as1 actparm, d) = transform-actparmlist(formparmlist, actparmlist)(dict) in
 6      let as1 tree = mk-Call-node1(as1 id, as1 actparm) in
 7      if d(IMPORTLIST) = ⟨⟩ then
 8        (⟨as1 tree⟩, d)
 9       else
10        (let (outputtrans, d') = transform-import(tnm, as1 tree, actl, term)(d) in
11         transform-transition(tnm, outputtrans, nodes1)(d')))
12    else
13     exit("§2.7.3: Length of procedure parameter list must equals the length of formal parameter list"))
```

type : [*Statename*$_0$] *Call*$_0$ *Actstmt*$_0$* [*Termstmt*$_0$] *Graph-node*$_1$* $\rightarrow$
Dict $\rightarrow$ (*Transition*$_1$ | *Graph-node*$_1$*) *Dict*

Objective Transform a procedure call into AS_1

Parameters

procid	The identifier of the procedure
actparmlist	The actual parameter list
tnm,	
actl,	
term,	
nodes$_1$	See *transform-act*

Result See *transform-act*

Algorithm

Line 1	Construct the *Qual* of the procedure.
Line 2	Let *formparmlist* denote the descriptors of the formal parameters and *nqual* denote the unique *Qual* (see the definition of *Newqual*).
Line 3	Construct the AS_1 identifier of the procedure.
Line 4	The length of the formal parameter list must be equal to the length of the actual parameter list.
Line 5	Transform the actual parameters.
Line 6	Construct the AS_1 call node.
Line 7-8	If there are no import expressions in the actual parameters then return the call node else
Line 10	Update *Statedict* to include the implicit import states and return the AS_0 transition leading to the implicit state.
Line 11	Transform the constructed transition

transform-actparmlist(*formparmlist*, *actparmlist*)(*dict*) $\triangleq$ (3.7.8.3.2)

```
1  (if formparmlist = ⟨⟩ then
2     (⟨⟩,dict)
3   else
4    (let (actparm, dict') =
5        if hd actparmlist = nil then
6          (⟨nil⟩, dict)
7         else
8          cases hd formparmlist:
9          (mk-InDescr(tqual)
10            → (let (as1 expr, , d) =
11                  transform-expr(hd actparmlist, EXPRESSION, {tqual})(dict) in
12               (as1 expr, d)),
13           mk-InoutDescr()
14            → transform-inoutactparm(hd formparmlist, hd actparmlist)(dict)) in
15    let (actparmrest, drest) =transform-actparmlist(tl formparmlist, tl actparmlist)(dict') in
16    (⟨actparm⟩ ⌢ actparmrest, drest)))
```

type: *FormparmD** *Actparmlist*$_0$ → *Dict* → [*Expression*$_1$]* *Dict*

Objective Transform an actual procedure parameter list into a list of AS_1 expressions

Parameters

formparmlist	The list of formal parameter descriptors
actparmlist	The list of actual parameters

Result The list of AS_1 expressions and a *Dict* possibly updated to include information of contained import expressions (see *transform-act*)

Algorithm

Line 1-2	When through, return the empty parameter list (the function is recursive).
Line 4-14	Let *actparm* denote the AS_1 expression denoting the first actual parameter.
Line 5-6	If the actual parameter is omitted then *actparm* is **nil** else
Line 9-12	If the parameter is an IN parameter then transform the expression which is the actual parameter. It must be of the sort denoted by *tqual*.
Line 13-14	If the parameter is an IN/OUT parameter then transform the IN/OUT parameter.
Line 15-15	Transform the rest of the actual parameters.
Line 16	Return the first actual parameter joined with the rest of the actual parameters

transform-inoutactparm(**mk-***InoutDescr*(*tqual*), *expr*)(*dict*) $\triangleq$ (3.7.8.3.3)

```
1  (if is-Id0(expr) then
2     (let vqual = get-visible-variable(expr, {get-parent(tqual)(dict)}, false)(dict) in
3      let mk-VarD(tq, , , , nqual) = dict(vqual) in
4      let as1 id = make-as1-identifier(nqual)(dict) in
5      if tq = tqual then
6        (as1 id, dict)
7       else
8        exit("§2.7.3: Same sort must be specified for formal and actual IN/OUT parameter"))
9    else
10    exit("§2.7.3: Actual IN/OUT parameter must be a variable identifier"))
```

type : *InoutDescr* $Expr_0 \rightarrow$ *Dict* $\rightarrow$ $Expression_1$ *Dict*

Objective Transform an actual IN/OUT parameter into AS_1

Parameters An IN/OUT descriptor containing

tqual	the *Qual* of its sort
expr	The actual parameter

Result The AS_1 actual parameter

Algorithm

Line 1	The actual parameter must be an identifier.
Line 2	Construct the *Qual* of the identifier.
Line 3	Let *tq* denote its sort or syntype and let *nqual* denote its *Newqual.*
Line 4	Construct its AS_1 identifier
Line 5-8	The sort of the variable (*tq*) must be equal to the sort of the formal parameter. In this case the same sort reference identifier must be specified, that is a sort is not compatible with syntypes of the sort.

3.7.8.4 Transformation of Decision Nodes

transform-decision(*tnm*, *decision*, *nodes*$_1$)(*dict*) $\triangleq$ (3.7.8.4.1)

```
1  (let mk-Decision0(quest, anslist, elsepart) = decision in
2   let isinformalanswers =
3       (∀mk-Condition0(v1, v2) ∈ elems anslist)(v1 = nil ∧ is-Stringterm0(v2) ∧ s-Qualifier0(v2) = ⟨⟩) in
4   let isinformalquestion = is-Stringterm0(quest) ∧ s-Qualifier0(quest) = ⟨⟩ in
5   let isinformaldecision = isinformalanswers ∧ isinformalquestion in
6   let qualset = all-visible-sorts(dict) in
7   let (,tpset,) = transform-expr(quest, EXPRESSION, qualset)(dict) in
8   let tp = if isinformaldecision then
9                nil
10              else
11               if isinformalquestion then
12                 (trap exit with get-answer-sort(anslist, qualset)(dict) in
13                  get-answer-sort(anslist, tpset)(dict))
14                else
15                 get-answer-sort(anslist, tpset)(dict) in
16  let (as1 trans, d'') = transform-answers(tnm, anslist, tp)(dict) in
17  let (els, d''') =
18      if elsepart = nil then
19        (nil, d'')
20       else
21        (let mk-Elsepart0(trans) = elsepart in
22         let (trans1, dict') = transform-transition(tnm, trans, ⟨⟩)(d'') in
23         (mk-Else-answer1(trans1), dict')) in
24  let (as1 quest',, d) =
25      if isinformalquestion then
26        (let mk-String0(str) = s-String0(quest) in
27         if isinformaldecision then
28           (mk-Informal-text1(str), {}, dict)
29          else
30           (trap exit with (mk-Informal-text1(str), {}, dict) in
31            transform-expr(quest, EXPRESSION, {tp})(d''')))
32       else
33        transform-expr(quest, EXPRESSION, {tp})(d''') in
34  let decisionnode1 = mk-Decision-node1(as1 quest', as1 trans, els) in
35  if d(IMPORTLIST) = ⟨⟩ then
36    (mk-Transition1(nodes1, decisionnode1), d)
37   else
38    (let (outputtrans, d') = transform-import(tnm, decisionnode1, ⟨⟩, nil)(d) in
39     transform-transition(tnm, outputtrans, nodes1)(d')))
```

type: [*Statename*$_0$] *Decision*$_0$ *Graph-node*$_1$* $\rightarrow$ *Dict* $\rightarrow$ *Transition*$_1$ *Dict*

Objective Transform a decision node into AS$_1$

Parameters

tnm,
decision,
nodes$_1$ — See *transform-act*. As opposed to the other action transforming functions, *transform-decision* does not take the action list following the decision as argument as a decision terminates a transition and because all answers at this stage contains terminators

Result A transition containing *nodes*$_1$ and either leading to the decision node or leading to an implicit import state

Algorithm

Line 1	Decompose the decision action.
Line 2	Let *isinformalanswers* be true if all answers consist of an unqualified character string.
Line 4	Let *isinformalquestion* be true if the expression in the question consist of an unqualified character string.
Line 5	Let *isinformaldecision* be true if the decision consist solely of informal text (character strings).
Line 6	Let *qualset* denote the *Quals* of all the sorts which are visible at this place.
Line 7	Extract the possible sorts for the decision question.
Line 8-15	Let *tp* denote the sort of the answers. If it is an informal decision then *tp* equals **nil** otherwise the sort is derived by looking at the answers and thereby restricting the set of sorts allowed for the question. Furthermore, if the question is a character string (line 11), then any visible sort may match the set of answers (line 12) if it fails to derive the answer sort from the set of sorts matching the question.
Line 16	Transform the answers given their sort (*tp* which is **nil** if the whole decision is informal).
Line 17-23	Transform the else part.
Line 21	Let *trans* denote the transition in the else part.
Line 22-23	Transform the transition and construct the AS_1 else part.
Line 24-33	Transform the question.
Line 24	If the question is an unquantified character string then it denotes informal text if the whole decision is informal (line 27) or if the transformation of the question as a character string literal fails given the answer sort *tp* (line 31).
Line 34	Construct the AS_1 decision node.
Line 35-36	If the question contains no import expressions then return the transition containing the preceding nodes and the decision node else
Line 38	Update *Statedict* to include the implicit import states and construct the AS_0 transition leading to the implicit import node.
Line 39	Transform the constructed transition

get-answer-sort(*anslist*, *tpset*)(*dict*) $\triangleq$ (3.7.8.4.2)

```
 1  (if anslist = ⟨⟩ then
 2     if card tpset ≠ 1 then
 3        exit("§2.2.2: No appropriate sort exist for decision action")
 4        else
 5        (let tp ∈ Qual be s.t. tp ∈ tpset in
 6         tp)
 7     else
 8     (let mk-Answer0(vlist, ) = hd anslist in
 9      let isinformal' =
10          len vlist = 1 ∧
11          (let mk-Condition0(v1, v2) = hd vlist in
12           v1 = nil ∧ is-Stringterm0(v2) ∧ s-Qualifier0(v2) = ⟨⟩) in
13      let (tset, ) = if isinformal' then (tpset, ) else transform-valueset(tpset, vlist)(dict) in
14      get-answer-sort(tl anslist, tset)(dict)))
```

type : $Answer_0{}^+$ *Sortqual*-**set** $\rightarrow$ *Dict* $\rightarrow$ [*Sortqual*]

Objective Extract the sort of a decision answers

Parameters

anslist	The list of answers attached to the decision action (excluding the elsepart)
tpset	The possible sorts of the answers. During each recursive call of *get-answer-sort* this set is restricted by the allowed sorts for the answer in hand. When initially applied, *tpset* denotes the possible sorts for the decision question or if the decision question is informal, all visible sorts.

Result The *Qual* of the sort for the decision action

Algorithm

Line 1-5	When through the answer list, the set of possible sorts must contain only one element and this element is returned.
Line 8	Decompose the first answer in the list.
Line 9-12	Let *isinformal'* be true if the answer consist of an unqualified character string.
Line 13	Let *tset* denote the set of sorts possible for the answers after having considered the first (next) answer. If the answer is an unqualified character string, it is not considered when determining the answer sort.
Line 14	Consider the rest of the answers.

transform-answers(*tnm*, *anslist*, *tp*)(*dict*) $\triangleq$ **(3.7.8.4.3)**

```
1  (if anslist = {} then
2     ({}, dict)
3   else
4    (let mk-Answer0(vlist, trans) = hd anslist in
5     let (,as1tree) =
6         if tp = nil then
7           ({}, nil)
8         else
9          (trap exit with ({},nil) in
10          transform-valueset({tp}, vlist)(dict)) in
11    let (trans1, dict') = transform-transition(tnm, trans, ⟨⟩)(dict) in
12    let (restrans, dict'') = transform-answers(tnm, tl anslist, tp)(dict') in
13    if as1tree = nil then
14      (let mk-Condition0(, mk-Stringterm0(, lit)) = hd vlist in
15       ({mk-Informal-text1(lit)} ∪ restrans, dict''))
16     else
17      ({mk-Decision-answer1(as1tree, trans1)} ∪ restrans, dict'')))
```

type: [*Statename*$_0$] *Answer*$_0$* [*Sortqual*] $\rightarrow$ *Dict* $\rightarrow$ *Decision-answer*$_1$-**set** *Dict*

Objective Transform the answers of a decision or option into AS_1

Parameters

tnm	See *transform-act*
anslist	The list of answers to be transformed
tp	The sort of the answers. If *tp* equals **nil** then the question and all answers consist of character strings (i.e. the decision is informal)

Result The set of AS_1 answers

Algorithm

Line 1	When through the answer list, return the empty set (the function is recursive).
Line 4	Let *vlist* and *trans* denote the conditions and the transition of the first answer in the list.

Line 5-10	Try to transform the conditions. If it fails then the conditions must contain informal text as *vlist* already has been checked for consistency (in *get-answer-sort*).
Line 11	Transform the transition of the first answer.
Line 12	Transform the rest of the answers.
Line 13-15	If the transformation of the condition list in hand failed then the condition list must denote informal text.
Line 15	Transform the condition as informal text and return the resulting answer joined with the rest of the AS_1 answers.
Line 17	Else return the transformed answer joined with the rest of the AS_1 answers

3.7.8.5 Transformation of Options

transform-option(*tnm*, **mk-***Option*$_0$(*quest*, *anslist*, *elsepart*), *nodes*$_1$)(*dict*) $\triangleq$ (3.7.8.5.1)

```
1  (let qualset = all-visible-sorts(dict) in
2   let (,tpset,) = transform-expr(quest, CONSTANT, qualset)(dict) in
3   let tp = get-answer-sort(anslist, tpset)(dict) in
4   let trans = eval-option-trans(tp, quest, anslist, elsepart)(dict) in
5   transform-transition(tnm, trans, nodes1)(dict))
```

type : [*Statename*$_0$] *Option*$_0$ *Graph-node*$_1$* $\rightarrow$ *Dict* $\rightarrow$ *Transition*$_1$ *Dict*

Objective Evaluate an option action and return the resulting transition

Parameters

tnm, *nodes*$_1$	See *transform-act*

Algorithm

Line 1	Let *qualset* denote the *Quals* of all the sorts which are visible at this place.
Line 2	Transform the option question. *tpset* denotes the sorts which are allowed for the question.
Line 3	Let *tp* denote the sort of the option. It is deduced by inspecting the answers (in the same way as done in *transform-decision*).
Line 4	Select the transition which matches the question.
Line 5	Transform the selected transition

eval-option-trans(*tp*, *question*, *anslist*, *elsepart*)(*dict*) $\triangleq$ (3.7.8.5.2)

```
1  (if anslist = ⟨⟩ then
2     if elsepart = nil then
3       exit("§4.3.4: No option answer matches the option question")
4      else
5       s-Transition0(elsepart)
6    else
7     (let mk-Answer0(vlist, trans) = hd anslist in
8      if match-option-answer(tp, vlist, question)(dict) then
9        if (∃ans ∈ elems tl anslist)(match-option-answer(tp, s-Conditionlist0(ans), question)(dict)) then
10         exit("§4.3.4: More than one option answer matches the question")
11       else
12        trans
13       else
14        eval-option-trans(tp, question, tl anslist, elsepart)(dict)))
```

type: *Sortqual Expr*$_0$ *Answer*$_0^*$ [*Elsepart*$_0$] $\rightarrow$ *Dict* $\rightarrow$ *Transition*$_0$

Objective Select the answer of an option action which matched the option question

Parameters

tp	The *Qual* of the question sort
question	The option question
anslist	The list of option answers
elsepart	The option else part

Result The transition contained in the matching answer

Algorithm

Line 1-5	If none of the answers matches the question then an else part must be specified and if so, return the transition contained in the else part.
Line 7	Decompose the first (the next) answer in the list.
Line 8	If it matches the question then
Line 9-12	It must be the only matching answer and if so, the transition contained in the answer is returned.
Line 14	Else continue searching for a matching answer

match-option-answer(*tp*, *vlist*, *question*)(*dict*) $\triangleq$ (3.7.8.5.3)

```
1  (let as0operator = build-answer-operator(tp, vlist, question) in
2   eval-simple-expr(as0operator, "BOOLEAN")(dict))
```

type: *Sortqual Conditionlist*$_0$ *Expr*$_0$ $\rightarrow$ *Dict* $\rightarrow$ *Bool*

Objective Test whether an option condition list matches the option question

Parameters

tp	The sort of the question and conditions
vlist	The condition list
question	The option question

Result True if the condition list matches the question

Algorithm

Line 1	From the question and the condition list, construct a boolean operator application which delivers true if the condition list matches the question.
Line 2	Evaluate the operator application

build-answer-operator(*tp*, *vlist*, *question*) $\triangleq$ (3.7.8.5.4)

```
1  (let mk-Condition0(op, const) = hd vlist in
2   let orop = mk-Qualop0(⟨⟩, mk-Quotedop0(OR)) in
3   let op' = if op = nil then EQ else op in
4   let as0op = if op' ∈ {NE, EQ, GT, LT, LE, GE} then
5                  mk-Operatorapp0(mk-Qualop0(tp, mk-Quotedop0(op')), ⟨question, const⟩)
6                else
7                 (let andop = mk-Qualop0(⟨⟩, mk-Quotedop0(AND)) in
8                  let lesseqop = mk-Qualop0(tp, mk-Quotedop0(LE)) in
9                  mk-Operatorapp0(andop, ⟨mk-Operatorapp0(lesseqop, ⟨op', question⟩),
10                                        mk-Operatorapp0(lesseqop, ⟨question, const⟩)⟩)) in
11  if tl vlist = ⟨⟩
12    then as0op
13    else mk-Operatorapp0(orop, ⟨as0op, build-answer-operator(tp, tl vlist, question)⟩))
```

type : *Sortqual Conditionlist*$_0$ *Expr*$_0$ $\rightarrow$ *Operatorapp*$_0$

Objective Construct a boolean operator which delivers the boolean literal true if an option answer matches the option question

Parameters

tp	The sort of the question
vlist	The condition list which is contained in the answer
question	The option question

Result The constructed AS_0 operator application

Algorithm

Line 1	Decompose the first condition in the condition list.
Line 2	Construct the prefix "OR" operator name.
Line 3	If no operator or expression is specified as the first element in the condition then it is the same as specifying the equality operator (EQ).
Line 4	If the first element in *Condition*$_0$ is an relational operator then construct the operator `tp op'(question,const)` where tp is the qualifier of the operator, op′ is the relational operator and (question,const) is the argument list.
Line 7-10	If the first element in *Condition*$_0$ is an expression then the condition reflects a range and the constructed operator is then `"AND"(tp "<="(op',question),tp "<="(question,const))` . where tp is the qualifier, op′ is the lower bound and const is the upper bound.
Line 11	If this was the last condition in the list then return the constructed operator application else
Line 13	Construct an operator application which "OR"s the result from the previous operator application with the result of the operator application constructed from the rest of the condition list

3.8 General AS_1 Creating Functions

make-as$_1$*-identifier*(*q*)(*dict*) $\triangleq$ **(3.8.1)**

```
1  (let qual = make-new-qual(q)(dict) in
2   let nameq = get-sur(qual) in
3   let (,nm) = qual[len qual] in
4   let as1 nameq = make-as1-qual(nameq) in
5   mk-Identifier1(as1 nameq, name-to-name1(nm)))
```

type: *Qual* → *Dict* → *Identifier*$_1$

Objective From a *Qual* (*q*) construct and return the corresponding AS_1 identifier

Algorithm

Line 1 Extract the *Newqual* (see the definition of *Newqual*) from the *Dict* descriptor, i.e. insert unique names in the *Qual*.

Line 2 Let *nameq* denote its qualifier (i.e. the *Qual* of the surrounding scopeunit).

Line 3 Extract the name part of the entity.

Line 4 Transform the *Qual* into an AS_1 qualifier.

Line 5 Construct the AS_1 identifier containing the AS_1 representation of the name, string or infix operator

make-as$_1$*-qual*(*qual*) $\triangleq$ **(3.8.2)**

```
1  if tl qual = ⟨⟩ then
2    ⟨⟩
3   else
4    (let (q, nm) = hd qual in
5     let as1 nm = name-to-name1(nm) in
6     let as1 q = cases q:
7                 (SYSTEM
8                   → mk-System-qualifier1(as1 nm),
9                  BLOCK
10                  → mk-Block-qualifier1(as1 nm),
11                 PROCESS
12                  → mk-Process-qualifier1(as1 nm),
13                 PROCEDURE
14                  → mk-Procedure-qualifier1(as1 nm),
15                 TYPE
16                  → mk-Sort-qualifier1(as1 nm),
17                 SIGNAL
18                  → mk-Signal-qualifier1(as1 nm),
19                 SUBSTRUCTURE
20                  → mk-Block-substructure-qualifier1(as1 nm)) in
21    let q' = make-as1-qual(tl qual) in
22    (as1 q ⌢ q'))
```

type: *Qual* → *Qualifier*$_1$

Objective Construct and return an AS_1 qualifier from a *Qual* (*qual*)

Algorithm

Line 1 When through, return the empty qualifier (the function is recursive).

Line 4 Let *q* and *nm* denote the entity class and the name of the first element in the qual.

Line 5	Let $as_1 nm$ denote the AS_1 representation of the name.
Line 7	If the entity class is the system then construct an AS_1 system qualifier element.
Line 9	If the entity class is a block then construct an AS_1 block qualifier element.
Line 11	If the entity class is a process then construct an AS_1 process qualifier element.
Line 13	If the entity class is a procedure then construct an AS_1 procedure qualifier element.
Line 15	If the entity class is sort then construct an AS_1 sort qualifier element.
Line 17	If the entity class is a signal then construct an AS_1 signal qualifier element.
Line 19	If the entity class is a block sub-structure then construct an AS_1 block sub-structure qualifier The two other entity classes services and channel sub-structure have been removed from the *Qual* in the function *make-new-qual.*
Line 21-22	Transform the rest of the *Qual* into AS_1 and join it with this element

make-new-qual(*qual*)(*dict*) $\triangleq$ **(3.8.3)**

```
 1  (let level = dict(SCOPEUNIT) in
 2   let qual' = cases dict(qual):
 3                 (mk-TimerD(, q)
 4                    → q,
 5                  mk-ProcedureD(, q)
 6                    → q,
 7                  mk-SortD(, , , q)
 8                    → q,
 9                  mk-SyntypeD(, q, , )
10                    → q,
11                  mk-VarD(, , , , q)
12                    → q,
13                  mk-OperatorD(, , q, )
14                    → q,
15                  mk-ChannelD(, , , , sub)
16                    → if sub = nil then qual else convert-channel-qual(sub, level),
17                  mk-ChannelsubD(bqual)
18                    → bqual ⌢ ⟨qual[len qual]⟩,
19                  mk-ServiceD(, , , , )
20                    → get-sur(qual),
21                  T → qual) in
22   let qualelem = qual'[len qual'] in
23   if tl qual' = ⟨⟩ then
24     ⟨qualelem⟩
25    else
26     (let qualrest =
27          if is-OperatorD(dict(qual)) ∨ is-LiteralD(dict(qual))
28            then get-sur(get-sur(qual'))
29            else get-sur(qual') in
30       make-new-qual(qualrest)(dict) ⌢ ⟨qualelem⟩))
```

type : *Qual* → *Dict* → *Qual*

Objective Insert unique names in the *Qual* of an entity. Unique names are necessary because of services and because of operator overloading

Result The new unique qual

Algorithm

Line 1	Let *level* denote the *Qual* of the scopeunit defining the entity
Line 3-21	In the descriptor of the entity, the new *Qual* is found. Only entities which may be defined in services, channels and channel sub-structure are renamed (channels are renamed if they contain a sub-structure (*sub* $\neq$ **nil**) and channel sub-structures are renamed to a block sub-structure). For the *Qual* of a service, the *Qual* of the enclosing process is returned.
Line 22-29	Take the *qual* of the scopeunit (*qualrest*) which defined the entity and rename that *Qual*. If the old *Qual* denotes an operator or a literal then do not use the TYPE qualifier element in the new *Qual*.
Line 30	Return the renamed scopeunit joined with the pair of entity class and name for the entity

convert-channel-qual(*sub*, *level*) $\triangleq$ (3.8.4)

```
1  (let (b1, c1, b2, c2) = sub in
2   if (∃q)(level = b1 ⌢ q) then
3     get-sur(b1) ⌢ ⟨(CHANNEL, c1)⟩
4    else
5     get-sur(b2) ⌢ ⟨(CHANNEL, c2)⟩)
```

type: *Newchannels Qual* $\rightarrow$ *Qual*

Objective Channels which contains a sub-structure are represented by two new channels, each leading to the sub-structure which is represented by a block sub-structure. When the name of the channel is used (in an identifier in a VIA set) it is replaced by the name of one of the new channels depending of where the channel is used. *convert-channel-qual* return the *Qual* of such a new channel. Note that it is not allowed to refer to the old channel name inside the channel sub-structure

Parameters

sub	The information about the two new channels. This information is extracted from the channel descriptor
level	The scopeunit where the channel is used

Result The *Qual* of the new channel

Algorithm

Line 1	Let *b1* denote one of the block endpoints, *c1* the new channel connected to that endpoint, *b2* the other endpoint and *c2* the new channel connected to that endpoint.
Line 2-5	If *level* indicates a scopeunit which is contained in *b1* then return the *Qual* of the channel connected to that block, else return the *Qual* of the other endpoint

make-as$_1$*idset*(*idset*)(*dict*) $\triangleq$ (3.8.5)

```
1  {make-as1-identifier(id)(dict) | id ∈ idset}
```

type: *Qual*-**set** $\rightarrow$ *Dict* $\rightarrow$ *Identifier*$_1$-**set**

Objective From a set of *Qual*s construct the corresponding set of AS_1 identifiers

make-as$_1$tree(*node*, *name*, *as$_1$set*, *parm1*, *parm2*, *parm3*)(*dict*) $\triangleq$ (3.8.6)

```
 1  (let as1name = name-to-name1(name),
 2       data = dict(DATATYPEDEF) in
 3   if sorts-have-values(data)(dict) then
 4     (let blocks = {elem | elem ∈ as1set ∧ is-Block-definition1(elem)},
 5          channels = {elem | elem ∈ as1set ∧ is-Channel-definition1(elem)},
 6          signals = {elem | elem ∈ as1set ∧ is-Signal-definition1(elem)},
 7          sigroutes = {elem | elem ∈ as1set ∧ is-Signal-route-definition1(elem)},
 8          syntypes = {elem | elem ∈ as1set ∧ is-Syn-type-definition1(elem)},
 9          procedures = {elem | elem ∈ as1set ∧ is-Procedure-definition1(elem)},
10          processs = {elem | elem ∈ as1set ∧ is-Process-definition1(elem)},
11          views = {elem | elem ∈ as1set ∧ is-View-definition1(elem)},
12          timers = {elem | elem ∈ as1set ∧ is-Timer-definition1(elem)},
13          vars = {elem | elem ∈ as1set ∧ is-Variable-definition1(elem)} in
14       cases node:
15        (SYSTEM
16           → mk-System-definition1(as1name, blocks, channels, signals, data, syntypes),
17         BLOCK
18           → mk-Block-definition1(as1name, processs, signals, parm1, sigroutes, data, syntypes, parm2),
19         SUBSTRUCTURE
20           → mk-Block-substructure-definition1(as1name, blocks, parm1, channels, signals, data, syntypes),
21         PROCESS
22           → mk-Process-definition1(as1name, parm1, parm2, procedures, signals, data,
23                                    syntypes, vars, views, timers, parm3),
24         PROCEDURE
25           → mk-Procedure-definition1(as1name, parm1, procedures, data, syntypes, vars, parm2)))
26     else
27      exit("§5.2.1: Sort in data type definition have no values"))
```

type : *Quot* [*Name$_0$*] *Decl$_1$*-**set** [*Channel-to-route-connection$_1$* | *Channel-connection$_1$* | *Number-of-instances$_1$* | *Procedure-formal-parameter$_1$**] [*Process-formal-parameter$_1$** | *Procedure-graph$_1$* | *Block-substructure-definition$_1$*] [*Process-graph$_1$*] → *Dict* → *Decl$_1$*

Objective Construct the AS$_1$ representation of a scopeunit, that is of a system, a block, a block sub-structure a process or a procedure.

Parameters

node	The kind of scopeunit to be constructed
name	The name of the scopeunit
as$_1$set	The set of definitions contained in the scopeunit
parm1,parm2,parm3	Some parameters which differs depending on the kind of scopeunit. For instance, if the scopeunit is a process then *parm1* is *Number-of-instances$_1$*

Result The AS$_1$ system, block, block sub-structure, process or procedure definition

Algorithm

Line 1	Extract the AS$_1$ representation of the name.
Line 2	Extract the *Data-type-definition$_1$* of the scopeunit.
Line 3	Every sort in the *Data-type-definition$_1$* must have at least one value.
Line 4-13	Partition the set of definitions into a set of block definitions.
Line 5	A set of channel definitions.
Line 6	A set of signal definitions.
Line 7	A set of signal route definitions.
Line 8	A set of syntype definitions.

Line 9	A set of procedure definitions.
Line 10	A set of process definitions.
Line 11	A set of view definitions.
Line 12	A set of timer definitions.
Line 13	A set of variable definitions.
Line 15-16	Construct a system definition.
Line 17-18	Construct a block definition.
Line 19-20	Construct a block sub-structure definition.
Line 21-23	Construct a process definition.
Line 24-25	Construct a procedure definition.

sorts-have-values(mk-*Data-type-definition*$_1$(, , *snmset*, *signatureset*,))(*dict*) $\triangleq$ (3.8.7)

```
1  (let level = dict(SCOPEUNIT) in
2   (∀nm ∈ snmset)
3    ((∃sign ∈ signatureset)
4      ((let mk-Identifier1(qual, signm) = s-Result1(sign) in
5        level = qual ∧ nm = signm))))
```

type: *Data-type-definition*$_1$ $\rightarrow$ *Dict* $\rightarrow$ *Bool*

Objective Test whether a data type definition for a scopeunit has values for every sorts i.e. whether there exist at least one literal of the sort or one operator which returns a value of the sort

Parameters The data type definition containing

snmset	The set of sorts defined by the data type definition
signatureset	The set of literal and operator signatures for the data type definition

Result True if success

Algorithm

Line 2	For all sort names in the set of sorts it must hold that there exist a signature in the set of signatures such that
Line 4-5	The result sort is locally defined and its name is the same as the sort name (*nm*)

$make\text{-}as_1\text{-}concaxioms(qualset)(dict) \triangleq$ **(3.8.8)**

```
 1  (if qualset = {} then
 2     {}
 3    else
 4     (let qual ∈ qualset in
 5      let (,nm) = qual[len qual] in
 6      let sort = get-sur(qual) in
 7      if is-Name0(nm) then
 8        make-as1-concaxioms(qualset \ {qual})(dict)
 9       else
10       (let mk-String0(str) = nm in
11        if len str ≤ 1 then
12          make-as1-concaxioms(qualset \ {qual})(dict)
13         else
14          (let ch = str[len str] in
15           let rest = mk-String0(⟨str[i] | 1 ≤ i ≤ len str − 1⟩) in
16           if sort ⌢ ⟨(LITERAL, rest)⟩ ∉ dom dict then
17             make-as1-concaxioms(qualset \ {qual})(dict)
18            else
19             (let concop = mk-Qualop0(sort, mk-Quotedop0(CONC)) in
20              let strexpr1 = mk-Stringterm0(⟨⟩, rest),
21                  strexpr2 = mk-Stringterm0(⟨⟩, mk-String0(⟨ch⟩)),
22                  strexpr3 = mk-Stringterm0(⟨⟩, nm) in
23              let equation =
24                  mk-Equation0(mk-Operatorterm0(concop, ⟨strexpr1, strexpr2⟩), strexpr3) in
25              let equation1 =
26                  (trap exit with {} in
27                   {transform-axiom(equation, AXIOMS)(dict)}) in
28              equation1 ∪ make-as1-concaxioms(qualset \ {qual})(dict))))))
```

type : $Qual\text{-}set \rightarrow Dict \rightarrow Equations_1$

Objective Construct the AS_1 representation of the implied concatenation axioms attached to character string literals, for example 'abc' have the implied axioms:

```
'ab'//'c' == 'abc';
'a'//'b'  == 'ab';
```

provided the equations are statically well-formed

Parameters The set of literal *Quals* defined in a partial data type definition

Result The constructed AS_1 equations

Algorithm

Line 1	When through, return the empty set of equations (the function is recursive).
Line 4	Consider the next literal.
Line 5	Let *nm* denote the name of the literal.
Line 6	Let *sort* denote its sort.
Line 7-8	If the literal is represented by a normal name then continue with the rest of the literals else
Line 10	Let *str* denote the characters in the string.
Line 11-12	Unless the number of character is greater than one then continue with the rest of the literals.
Line 14	Let *ch* denote the last character in the string.
Line 15	Let *rest* denote the string containing all but the last character.

Line 16-17	If the string where the last character has been removed is not defined for the sort then continue with the rest of the literals.
Line 19	Construct the "//" operator.
Line 20	Construct a string term containing the rest of the characters and containing no qualifier (*strexpr1*).
Line 21	Construct a string term containing the last character and containing no qualifier (*strexpr2*).
Line 22	Construct a string term containing the string literal (*strexpr3*).
Line 23	Construct the equation : `strexpr1 // strexpr2 == strexpr3`
Line 25-27	Try to transform the equation into AS_1. If it fails then the equation is not well-formed and is therefore not implied.
Line 28	Join the equation with the equations implied from the rest of the literals

$as_0\text{-}id(qual) \triangleq$ **(3.8.9)**

```
1  if qual = ENV then
2    ENV
3   else
4    (let (,nm) = qual[len qual] in
5     let qualifier = get-sur(qual) in
6     mk-Id0(qualifier, nm))
```

type : $(Qual \mid \mathsf{ENV}) \rightarrow (Id_0 \mid \mathsf{ENV})$

Objective Retrieve an AS_0 identifier from a *Qual*. This function is used in the expansion of short-hands.

Parameters

qual	The *Qual* to be converted to an AS_0 identifier. As the function also is applied channel and signal route endpoints, the function may take ENV as argument.

Result The constructed AS_0 identifier

Algorithm

Line 1	If it is ENV then do not change it else
Line 4	Extract the name part.
Line 5	Extract the qualifier which is the *Qual* of the surrounding scopeunit.
Line 6	Return the composed identifier

make-implicit-vardecl(dict) $\triangleq$ **(3.8.10)**

```
1  (let (,f1nm,f2nm) = dict(GLOBALNAMES) in
2   let as0id = as0-id(get-predef-sort("INTEGER")(dict)) in
3   let init = mk-Id0(⟨⟩, mk-Name0("0", nil)) in
4   let formudecl = mk-Vardef0(nil, nil, ⟨mk-Vardefelem0(⟨f1nm,f2nm⟩, as0id, init)⟩) in
5   let closureset = dict(IMPLIED) in
6   let as0defs = formudecl ⌢ ⟨make-implicit-import-vardef(closure) | closure ∈ closureset⟩ in
7   transform-vardef(as0defs)(dict))
```

type : *Dict* → *Variable-definition*$_1$-**set** *Dict*

Objective Construct the AS_1 variable definitions for the two implicit variables used in enabling condition and continuous signal and construct the AS_1 variable definitions for the implicit import variables in a process

Algorithm

Line 1	Let *f1nm* and *f2nm* denote the AS_0 names of the two implicit variables used in enabling condition and continuous signal.
Line 2	Let as_0id denote the AS_0 identifier of the INTEGER sort.
Line 3	Construct the initiation expression. The variables are initialized to zero.
Line 4	Construct the AS_0 variable definitions for the two variables.
Line 5	Extract the set of pairs of variable name and variable sort containing the implicit variables used in the transformation of the import expressions.
Line 6	Let as_0defs denote all the implicit definitions.
Line 7	Transform the definitions into AS_1

$make\text{-}implicit\text{-}import\text{-}vardef((t, nm)) \triangleq$ (3.8.11)

1 (**let** $as_0tid = as_0\text{-}id(t)$ **in**
2 **mk-**$Vardef_0$(**nil**, **nil**, ⟨**mk-**$Vardefelem_0$(⟨nm⟩, as_0tid, **nil**)⟩))

type: $NameclosureD \rightarrow Vardef_0$

Objective Construct the AS_0 variable definition for an implicit import variable

Parameters The pair of variable sort t and variable name (nm)

Line 1	Construct the AS_0 identifier of the variable sort.
Line 2	Construct the AS_0 variable definition. The variable is not initialized as the first use of it always is in an input node

transform-import(*tnm*, *originalnode*$_1$, *actl*, *term*)(*dict*) $\triangleq$ (3.8.12)

```
 1  (let importlist = dict(IMPORTLIST) in
 2   let (impnm, qual, expr) = hd importlist,
 3       statenm = create-unique-name() in
 4   let (, (nm, )) = qual[len qual] in
 5   let mk-ImportD(tq) = dict(qual) in
 6   let mk-SignalnamesD(, xq, xr) = exportmap((tq, nm)) in
 7   let lev = dict(SCOPEUNIT) in
 8   let osig = mk-Outputsig0(mk-Id0(⟨⟩, xq), ⟨⟩) in
 9   let outputnode = mk-Actstmt0(nil, mk-Output0(⟨osig⟩, expr, ⟨⟩)) in
10   let sigset = all-input-signals(lev)(dict) \ {lev ↷ ⟨(SIGNAL, xr)⟩} in
11   let savenode = mk-Savespec0(⟨as0-id(sig) | sig ∈ sigset⟩),
12       statedict = dict(STATEDICT) in
13   let termstmt' = mk-Termstmt0(nil, mk-Nextstate0(mk-Id0(⟨⟩, statenm))) in
14   let outputtrans = mk-Transition0(outputnode, termstmt') in
15   let impdict = [IMPLIED      ↦ (dict(IMPLIED) ∪ {(tq, impnm)}),
16                  IMPORTLIST ↦ tl importlist] in
17   if tl importlist = ⟨⟩ then
18     (let inputtrans = mk-Transition0(actl, term) in
19      let inpvars = ⟨mk-Inputvars0(mk-Id0(⟨⟩, xr), ⟨mk-Id0(⟨⟩, impnm)⟩)⟩ in
20      let inputnode = mk-Inputspec0(inpvars, nil, inputtrans) in
21      let stated = mk-StateD(⟨(lev, inputnode), (lev, savenode)⟩, (tnm, originalnode1)) in
22      let statedict' = statedict + [statenm ↦ stated] in
23      let dict' = dict + [STATEDICT ↦ statedict'] in
24      (outputtrans, dict' + impdict))
25    else
26     (let (termstmt'', dict'') = transform-import(tnm, originalnode1, actl, term)(dict + impdict) in
27      let inpvars = ⟨mk-Inputvars0(mk-Id0(⟨⟩, xr), ⟨mk-Id0(⟨⟩, impnm)⟩)⟩ in
28      let inputnode = mk-Inputspec0(inpvars, nil, termstmt'') in
29      let statedict' =
30          statedict + [statenm ↦ mk-StateD(⟨(lev, inputnode), (lev, savenode)⟩, (tnm, nil))] in
31      let dict''' = dict'' + [STATEDICT ↦ statedict'] in
32      (outputtrans, dict''')))
```

type: *Statename*$_0$ (*Graph-node*$_1$ | *Decision-node*$_1$) *Actstmt*$_0^+$ [*Termstmt*$_0$] $\rightarrow$ *Dict* $\rightarrow$ *Transition*$_0$ *Dict*

Objective Update *Statedict* in *Dict* to include the import states implied from an expression and construct the transition leading to the implicit state

Parameters

tnm — The name of the state to which the action containing the expression belongs. The name is saved in the *StateD* descriptor of the import state such that during the transformation of the transition following the import state, dashed nextstate can be properly substituted

originalnode$_1$ — The AS$_1$ version of the action which contained the import expression. This node is also saved in the import state (the two objects forms the *Importstateinf* defined in *Quotdict*) as it must be included in the AS$_1$ transition following the import state

actl,term — The AS$_0$ actions and terminator which followed the action containing import expression(s). The transition which follows the xtREPLY input contains these two objects. When the import state has been transformed, *originalnode*$_1$ is added to the transition (see *insert-importact*). If the action contained more than one import expression then the transition following the xtREPLY input for the other import states consist of an output of the xtQUERY signal and a nextstate to the next import state.

Result An AS_0 transition containing an output of the xtQUERY signal associated to the first import state and a *Dict* updated to include descriptors for the implicit import states

Algorithm

Line 1 Extract the information about the import expressions which occurred in the expression.

Line 2 Extract information about the first import expression. *impnm* denotes the name of the implicit variable replacing the import expression, *qual* denotes the *Qual* of the import variable used in the import expression and *expr* denotes the AS_0 pid expression from the import expression.

Line 3 Construct a name for the implicit state.

Line 4 Extract the name of the import variable.

Line 5 Extract the *Qual* of the result sort of the import expression.

Line 6 Decompose the *SignalnamesD* associated to the pair of result sort and import variable name. *xq* is the name of the xtQUERY signal and *xr* is the name of the xtREPLY signal.

Line 7 Let *lev* denote the current scopeunit, that is the process, a service or a procedure.

Line 8-9 Construct the output action containing the xtQUERY signal and the PiD expression.

Line 10 Let *sigset* denote the *Quals* of the signals which must be saved in the implicit state.

Line 11 Construct the AS_0 save node.

Line 13-14 Construct the transition containing the output of the xtQUERY signal and which terminates in the implicit state.

Line 15 Add the new import variable to the set of implicit variables contained in the IMPLIED entry in *Dict* and remove the information about the import expression from the list contained in the IMPORTLIST entry.

Line 17 If this import expression was the last (or the only one) in the expression then

Line 18 Construct the transition containing the actions and terminator which followed the action containing the expression.

Line 20 Construct the input of the xtREPLY signal.

Line 22-23 Add the implicit state to *Statedict*. *lev* indicates that the input node and the save node shall be transformed in the same context as the one in which the import expression occurred (recall that the states are transformed after services have been merged).

Line 24 Return the transition containing the output and return the updated *Dict*.

Line 26 If there are more import expressions then construct the output transition leading to the next import state and update *Dict* to include the rest of the import states and to include the new implicit variable.

Line 28 Construct the input containing the transition leading to the next import state.

Line 29-31 Update *Statedict* to include the new implicit state. The *Importstateinf* descriptor in the *StateD* descriptor do not contain the AS_1 action ($originalnode_1$) since this state is not the last of the states implied by the import expressions in the action.

Line 32 Return the transition containing the output and return the updated *Dict*

3.9 Expansion of Services

build-service-descriptor(**mk-***Servicedef*$_0$(*id*, *sigset*, *dcll*, *body*, *tid*))(*dict*) $\triangleq$ **(3.9.1)**

```
1  (let mk-Id0(q, snm) = id in
2   let squal = dict(SCOPEUNIT) ⌢ ⟨(SERVICE, snm)⟩ in
3   let dict' = dict + [SCOPEUNIT ↦ squal] in
4   if q ≠ ⟨⟩ then
5     exit("§2.4.1: Defining names may only be qualified in remote definitions")
6    else
7     if tid ∈ {id, nil} then
8       (let sigqualset = transform-validinputset(sigset)(dict') in
9        let (as1 dclset, dict'') = transform-decllist(dcll)(dict') in
10       let (starttrans, labeldict, statedict, priinput) =
11           build-service-statedict(body)(dict) in
12       (as1 dclset, dict'' + [squal ↦ mk-ServiceD(starttrans, labeldict, statedict, sigqualset, priinput)]))
13      else
14       exit("§2.2.2: Ending name in service definition is different from defining name"))
```

type: *Servicedef*$_0$ $\rightarrow$ *Dict* $\rightarrow$ *Decl*$_1$-**set** *Dict*

Objective Transform the definitions contained in a service definition into AS_1 and construct a service descriptor which contains the service graph in the form of a *Statedict* and a *Labeldict*

Parameters The service definition containing

id	The identifier of the service
sigset	The valid input signal set for the service
dcll	The local definitions
body	The service graph
tid	The identifier ending the service definition

Result The AS_1 definitions of the local definitions and a *Dict* containing the service descriptor and the descriptors of the local definitions

Algorithm

Line 2-3	Update SCOPEUNIT in *Dict* to denote the service
Line 4-7	The service identifier must not be qualified (line 4) and if the ending identifier is specified then it must be the same as the service identifier (line 7)
Line 8	The signals which may be received in an input or a priority input in the service are the signals specified in the valid input signal set or specified in the service signal route joined with the timers defined in the service.
Line 9	Transform the local definitions.
Line 10	Extract the initial transition (*starttrans*), construct the *Statedict* (*statedict*) and the *Labeldict* (*labeldict*) from the service graph and extract the signals which are high priority.
Line 12	Return the AS_1 version of the local definitions and the *Dict* updated to include their descriptors and the descriptor of the service

build-service-statedict(**mk-***Body*$_0$(*trans*, *statelist*))(*dict*) $\triangleq$ (3.9.2)

```
1  (let trans' = insert-trans-term(trans) in
2   let statelist' = ⟨insert-state-term(statelist[i]) | 1 ≤ i ≤ len statelist⟩ in
3   let labeldict = build-trans-labeldict(trans')([]) in
4   let labeldict' = build-state-labeldict(statelist')(labeldict) in
5   let statedict = build-statedict(statelist', dict(SCOPEUNIT))([]) in
6   let statedict' = remove-asterisk-input-and-save(statedict)(dict) in
7   let priinput = extract-priinput(statedict')(dict) in
8   (trans', labeldict', statedict', priinput))
```

type : *Body*$_0$ → *Dict* → *Transition*$_0$ *Labeldict Statedict Signalqual*-**set**

Objective Construct the *Statedict* and the *Labeldict* for a service

Parameters The body of a service containing

trans	The initial transition
statelist	The states of the service

Result The initial transition where terminators have been inserted in decision answers, the constructed *Labeldict*, the constructed *Statedict* and the *Quals* of the high priority signals

Algorithm

Line 1	Insert terminators in the initial transition.
Line 2	Insert terminators in the states of the service.
Line 3	Construct the *Labeldict* for the initial transition.
Line 4	Extend the *Labeldict* to include the whole service body.
Line 5	Construct the *Statedict*.
Line 6	Remove the asterisk inputs and asterisk saves from the states in *Statedict*.
Line 7	Extract the *Quals* of the high priority signals.
Line 8	Return the initial transition, the *Labeldict*, the *Statedict* and the high priority signals

extract-priinput(*statedict*)(*dict*) $\triangleq$ (3.9.3)

```
1  (if statedict = [] then
2     {}
3   else
4    (let stnm ∈ dom statedict in
5     let mk-StateD(speclist, ) = statedict(stnm) in
6     let inputset = {inp | (, inp) ∈ elems speclist ∧ is-Priinput0(inp)} in
7     let inpvars = {inpv | (∃mk-Priinput0(inpl, ) ∈ inputset)(inpv ∈ elems inpl)} in
8     let priset = {signal-qual(id)(dict) |
9                   (∃mk-Inputvars0(id', ) ∈ inpvars)(id' = id)} in
10    priset ∪ extract-priinput(statedict \ {stnm})(dict)))
```

type : *Statedict* → *Dict* → *Signalqual*-**set**

Objective Extract the high priority signals from the states of a service by inspecting the input nodes

Parameters The *Statedict* containing the service states

Result The high priority signals of the service

Algorithm

Line 1	When through statedict, return the empty set (the function is recursive).

Line 4	Take some state in *Statedict*.
Line 5	Extract the inputs of the state.
Line 6	Extract the priority input nodes from the state.
Line 7-8	Extract the high priority signals from the priority inputs.
Line 10	Return the high priority signals together with the high priority signals from the rest of the state

transform-decomposition-body(*dict*) $\triangleq$ (3.9.4)

```
1  (let servicelist = ⟨squal ∈ dom dict | is-local(squal)(dict) ∧
2                                          is-ServiceD(dict(squal))⟩ in
3   if servicelist = ⟨⟩ then
4     exit("§4.10: A decomposition must contain at least one service definition")
5    else
6     (let statetuplemap = merge-states(servicelist, ⟨⟩, ⟨⟩, 1)(dict) in
7      let statetuplenamemap = [statetuple ↦ create-unique-name() |
8                                  statetuple ∈ dom statetuplemap] in
9      let statedict = [statetuplenamemap(tuple) ↦ (statetuplemap(tuple), nil, nil) |
10                      tuple ∈ dom statetuplemap] in
11     let statedict' = double-states(statedict, servicelist)(dict) in
12     let statedict'' = remove-cont-enable-from-statelist(dom statedict')(dict, statedict') in
13     let dict' = dict + [SERVICES   ↦ (statetuplenamemap, servicelist),
14                         STATEDICT ↦ statedict''] in
15     let (firststate, firstservice) = extract-firststate-or-service(servicelist, ⟨⟩, nil)(dict) in
16     if firstservice = nil then
17       (let term1 = mk-Nextstate-node1(name-to-name1(firststate)) in
18        let trans1 = add-service-varinit(mk-Transition1(⟨⟩, term1), elems servicelist)(dict) in
19        let (statebody1, dict'''') = transform-statelist({})(dict') in
20        (mk-Process-graph1(mk-Process-start-node1(trans1), statebody1), dict''''))
21      else
22       (let mk-ServiceD(trans, labeldict, , , ) = dict(firstservice) in
23        let dict'' = dict' + [LABELDICT ↦ labeldict,
24                              SCOPEUNIT ↦ firstservice] in
25        let (trans1, dict''') = transform-transition(firststate, trans, ⟨⟩)(dict'') in
26        let trans1' = add-service-varinit(trans1, elems servicelist)(dict) in
27        let (statebody1, dict'''') = transform-statelist({})(dict''') in
28        (mk-Process-graph1(mk-Process-start-node1(trans1'), statebody1), dict'''')))
```

type: *Dict* → *Process-graph*$_1$ *Dict*

Objective Transform a service decomposition into an AS_1 process graph

Parameters The *Dict* which contains the descriptors of the services

Result The AS_1 process graph and a *Dict* which contains information about output signals used in the services and about implicit variables (same as for *transform-body*)

Algorithm

Line 1-2	Create a list of service *Quals*. This list is essential in the transformation of services. The positions in the list correspond to the positions in the state name tuples as also mentioned in the transformation model in §4.10.2 of Z.100.
Line 3	There must exist at least one service in a decomposition
Line 6	Merge the states of the services. The result is a special *Statedict* where the entries are state name tuples instead of state names.
Line 7-8	Construct a map which defines a unique state name for each state name tuple in the special *Statedict*.

Line 9	Construct a normal *Statedict* where the name tuples are replaced by the associated unique names.
Line 11	Construct a new *Statedict* where each state is split into two states: The state receiving the priority inputs and the state receiving all other inputs. The second of these states are given a unique name.
Line 12	Remove enabling conditions and continuous signals from the *Statedict* in the same way as done for a normal process graph.
Line 13	Make the servicelist and the relation between the state name tuples and the unique state names available for the state transforming functions by including the information in *Dict*. Also include the *Statedict* in *Dict* in the same way as done for a normal process graph.
Line 15	Extract the *Qual* of the service which contains the initial actions or extract the first state in the case where there is no service which contains an initial transition i.e. either *firststate* is **nil** or *firstservice* is **nil**.
Line 16	If there is no service which contains an initial transition then
Line 17	Construct the AS_1 nextstate to the first state.
Line 18	Construct the AS_1 action list which initiates the variables of the services and which terminates with the nextstate to the first state.
Line 19	Transform the states.
Line 20	Return the process graph where the initial transition contains the initiation of the variables and the constructed nextstate node.
Line 22	If there is a service containing an initial transition then extract it (*trans*). Also extract the *Labeldict* to be used in the transformation of the initial transition.
Line 23	Make *Labeldict* and the context in which the initial transition is transformed in, available for the transformation function by including them in *Dict*.
Line 25	Transform the initial transition.
Line 26	Add the AS_1 action list which initiates the variables of the services to the initial transition.
Line 27	Transform the states.
Line 28	Return the process graph

double-states(*statedict*, *servicelist*)(*dict*) $\triangleq$ (3.9.5)

```
 1  (if statedict = [] then
 2     []
 3   else
 4    (let stnm ∈ dom statedict in
 5     let mk-StateD(speclist, ) = statedict(stnm) in
 6     let (priinput, norminput) be s.t. priinput ∪ norminput = elems speclist ∧
 7         (∀(,inp) ∈ priinput)(is-Priinput0(inp)) ∧
 8         (∀(,inp) ∈ norminput)(¬is-Priinput0(inp)) in
 9     if priinput = {} then
10       [stnm ↦ statedict(stnm)] + double-states(statedict \ {stnm}, servicelist)(dict)
11      else
12       (let saveset =
13           {(servicelist[i], mk-Savespec0(siglist)) | i ∈ ind servicelist ∧
14                                                 (let sigset =
15                                                      all-input-signals(servicelist[i])(dict) in
16                                                  siglist = ⟨as0-id(sig) | sig ∈ sigset⟩)} in
17        let qual ∈ elems servicelist in
18        let newstnm = create-unique-name() in
19        let continput = (qual, mk-Contspec0(mk-Id0(⟨⟩, mk-Name0("TRUE", nil)), nil,
20                                            mk-Termstmt0(nil, mk-Nextstate0(newstnm)))) in
21        let firstspec = ⟨spec | spec ∈ (priinput ∪ saveset)⟩ ⌢ ⟨(continput, nil)⟩,
22            secondspec = ⟨spec | spec ∈ norminput⟩ in
23        let firststate = [stnm ↦ mk-StateD(firstspec, nil)],
24            secondstate = [newstnm ↦ mk-StateD(secondspec, (stnm, nil))] in
25        firststate + secondstate + double-states(statedict \ {stnm}, servicelist)(dict))))
```

type: *Statedict Servicequal*$^+$ $\rightarrow$ *Dict* $\rightarrow$ *Statedict*

Objective Replace every state in *Statedict* by two states where the first state has the priority signals as inputs and the second state has the other signals as inputs.

Parameters

statedict	The (rest of) the old *Statedict* (the function is recursive)
servicelist	The list of service *Quals* (see *transform-decomposition*)

Result The expanded *Statedict*

Algorithm

Line 1	When through, return the empty map.
Line 4	Take the next state to be dealt with.
Line 5	Let *speclist* denote the list of inputs.
Line 6-8	Split the inputs into priority inputs and other inputs.
Line 9-10	If there is no priority inputs then do not modify the state.
Line 12-16	Construct the *Specs* which each contains a save of all the normal signals. There is one *Spec* for each service.
Line 17	Take some arbitrary service *Qual* from the service list. This service is used in the constructed *Spec* (line 19) which contains the continuous signal to the second state, that is, the constructed continuous signal is transformed in the context of an arbitrary service because the continuous signal do not use any identifiers which are context dependent.
Line 18	Create a unique name for the second state.
Line 19	Construct the continuous signal which leads to the second state. The condition in the continuous signal is "true". Note that continuous signal has not been mentioned explicit in the model of services

in Z.100. However, SAMETOKEN (see §4.11 of Z.100) correspond to the implicit variable (called "n" in §4.12) and XCONT correspond to the implicit signal emptyq.

Line 21 Construct the *Speclist* for the first state. It includes the continuous signal.

Line 22 Construct the *Speclist* for the second state.

Line 23 Construct the *Statedict* contribution for the first state.

Line 24 Construct the *Statedict* contribution for the second state.

Line 25 Continue to double the rest of the states

extract-firststate-or-service(*servicelist*, *statelist*, *firstservice*)(*dict*) $\triangleq$ **(3.9.6)**

```
1  (if servicelist = ⟨⟩ then
2     if firstservice ≠ nil then
3       (nil, firstservice)
4      else
5       (let (statetuplenamemap, ) = dict(SERVICES) in
6        (statetuplenamemap(statelist), firstservice))
7    else
8     (let mk-ServiceD(mk-Transition0(actl, term), , stdict, , ) = dict(hd servicelist) in
9      if actl = ⟨⟩ ∧ is-Nextstate0(s-Terminator0(term)) then
10       (let mk-Termstmt0(, mk-Nextstate0(stnm)) = term in
11        if stnm ∈ dom stdict then
12          extract-firststate-or-service(tl servicelist, statelist ⌢ ⟨stnm⟩, firstservice)(dict)
13         else
14          exit("§2.6.7.2.1: Name in nextstate must denote a defined state"))
15      else
16       if firstservice ≠ nil then
17         exit("§4.10.2: More than one service contains an initial transition string")
18        else
19         extract-firststate-or-service(tl servicelist, nil, hd servicelist)(dict)))
```

type : *Servicequal** [*Statename*$_0$*] [*Servicequal*] $\rightarrow$ *Dict* $\rightarrow$ [*Statename*$_0$] [*Servicequal*]

Objective Extract either the initial state of the composite states in *Statedict* or the service which contains an initial transition.

Parameters

servicelist The list of services (see *transform-decomposition*)

statelist A state name tuple which is constructed during the recursive call of the function. When all the services in *servicelist* have been considered, this name tuple defines the unique state name to be returned (via the state tuple map, see *transform-decomposition*)

firstservice If a service containing an initial transition has been found then *firstservice* contains the *Qual* of that service. As soon as such a service has been found, the recursion could stop and the service returned, but for the sake of checking that there is only one such service, the service is taken as parameter to the next recursion call

Result If there exist a service which contains an initial transition then the *Qual* of that service (*Servicequal*) else the name of the unique starting state

Algorithm

Line 1-2 When having considered all the services then if a service containing an initial transition has been found then return the *Qual* of that service (*firstservice*) else

Line 5-6 the starting state is the state which correspond to the initial next-states in the services (*statelist* contains those initial service states).

Line 8	Decompose the descriptor of the next service to be considered.
Line 9	If the start transition contains a nextstate only then
Line 10	Let *stnm* denote the name of the state in the nextstate.
Line 11	If the state is defined in the service (i.e. if it can be found in the *Statedict* local to the service) then
Line 12	Continue with considering the next service. Add the state name to the state name list such that eventually the new unique state can be found if none of the services contains an initial transition string (line 6).
Line 16-19	If a service containing an initial transition string has not already been found then consider the next service. As a service containing an initial transition has been found the state name list is of no use (i.e. it is **nil**)

add-service-varinit(*trans*, *squalset*)(*dict*) $\triangleq$ **(3.9.7)**

```
1  (if squalset = {} then
2     add-varinit(trans, dict)
3    else
4     (let squal ∈ squalset in
5      let dict' = dict + [SCOPEUNIT ↦ squal] in
6      let trans' = add-varinit(trans, dict') in
7      add-service-varinit(trans', squalset \ {squal})(dict)))
```

type: *Transition*$_1$ *Servicequal*-**set** $\rightarrow$ *Dict* $\rightarrow$ *Transition*$_1$

Objective Add the AS_1 initiations of the service variables to the initial transition.

Parameters

trans	The initial transition
squalset	The *Quals* of the services of a service decomposition

Result The updated transition

Algorithm

Line 1	When through all the services then add the initiations of the variables defined on process level to the transition.
Line 4	Let *squal* denote one of the (remaining) services.
Line 5	Let SCOPEUNIT in *Dict* denote that service.
Line 6	Let *trans'* denote the transition updated the variable initiations for that service. The function *add-varinit* is also used when constructing the initial transition attached to a process or procedure.
Line 7	Add the initiations for the remaining services

merge-states(*servicelist*, *statenmtuple*, *statelist*, *index*)(*dict*) $\triangleq$ **(3.9.8)**

```
1  (if index > len servicelist then
2     [statenmtuple ↦ mk-StateD(statelist, nil)]
3    else
4     (let mk-ServiceD(, , statedict, , ) = dict(servicelist[index]) in
5      merge {merge-states(servicelist,
6                          statenmtuple ⌢ ⟨statenm⟩,
7                          statelist ⌢ s-Speclist(statedict(statenm)),
8                          index + 1)(dict) | statenm ∈ dom statedict}))
```

type: *Servicequal*$^+$ *Statename*$_0$* *Speclist* N_1 $\rightarrow$ *Dict* $\rightarrow$ (*Statename*$_0$$^+$ $\overrightarrow{m}$ *StateD*)

Objective Merge ("STATE EXPLODE") the *Statedicts* of the services into a *Statedict* like map where the entries are state name tuples (same kind of state name tuples as defined in the model-part of §4.10.2 of Z.100). The function *transform-decomposition-body*, where this function is applied, introduces a unique state name for each of the tuples and thereby converts the map into an ordinary *Statedict*

Parameters

servicelist The list of *Servicequals* of the services. The positions of a service in this list equals the position of its states in the state name tuples

statenmtuple, statelist The state name tuple (*statenmtuple*) and the inputs (*statelist*) for a resulting state is creating recursively. When *merge-states* initially is applied, they are empty.

index The index to *servicelist* which indicates which service to be treated at this recursion level.

Algorithm

Line 1 When having constructed a (complete) state name tuple (*statenmtuple*) then return the contribution for this particular tuple.

Line 4 Let *statedict* denote the *Statedict* of the service corresponding to *index*.

Line 5-9 For this service, form the map which includes all the state tuples which have *statenmtuple* as the first elements (formed from the services which already have been treated) and which have a state (*statenm*) from the service as the next element. The contributions from the various states of the service are unified into a single map by means of the **merge** operator

$$is\text{-}wf\text{-}servicesignals(dict) \triangleq \tag{3.9.9}$$

```
1  (let processsignals =
2      {squal | (∃mk-SignalrouteD(p1, p2, s1, s2) ∈ rng dict)
3               ((p1 = dict(SCOPEUNIT) ∧ squal ∈ s2) ∨ (p2 = dict(SCOPEUNIT) ∧ squal ∈ s1))} in
4   (∃d1, d2 ∈ dom dict)(is-ServiceD(dict(d1)) ∧ is-ServiceD(dict(d2)) ∧
5                        get-sur(d1) = dict(SCOPEUNIT) ∧ get-sur(d2) = dict(SCOPEUNIT) ∧
6                        s-Validinputset(dict(d1)) ∩ s-Validinputset(dict(d2)) ≠ {} ⊃
7                        (d1 = d2 ∧ s-Priinputset(dict(d1)) ∩ processsignals = {})))
```

type: $Dict \rightarrow Bool$

Objective Check that the input signals of the services are disjoint and that no priority signals are external to the enclosing process

Result True if the signals are well-formed

Algorithm

Line 1-3 Let *processsignals* denote the signals contained in a signal route (implicit or explicit).

Line 4 For every two *Quals d1* and *d2* of entities defined in a process it must hold that if they denote services (line 4) and they are defined in the surrounding process (line 5) and they have input signals in common (line 6) then they must denote the same service and that service must have no high priority signals leading to the environment of the surrounding process (line 7).

transform-servicesigroutedef(mk-*Sigroutedef*$_0$(*nm*, *path*, *opath*))(*dict*) $\triangleq$ (3.9.10)

```
1  (let mk-Sigroutepath0(endpoint1, endpoint2, siglist) = path in
2   let sigroutequal = dict(SCOPEUNIT) ↷ ⟨(SIGNALROUTE, nm)⟩ in
3   let sigset = transform-signallist(siglist)(dict) in
4   let s1 = get-visible-qual(endpoint1, SERVICE)(dict),
5       s2 = get-visible-qual(endpoint2, SERVICE)(dict) in
6   let osigset =
7       if opath = nil then
8         {}
9        else
10        (let mk-Sigroutepath0(orig', dest', osiglist) = opath in
11         let s1' = get-visible-qual(orig', SERVICE)(dict),
12             s2' = get-visible-qual(dest', SERVICE)(dict) in
13         if s1 = s2' ∧ s2 = s1' then
14           transform-signallist(osiglist)(dict)
15          else
16           exit("§4.10.1: The endpoints in the directions of a bidirectional signal route must match")) in
17  if s1 ≠ s2 then
18    (let descr = mk-SignalrouteD(s1, s2, sigset, osigset) in
19     ({}, [sigroutequal ↦ descr]))
20   else
21    exit("§4.10.1: The endpoints of service signal route must be different"))
```

type: *Sigroutedef*$_0$ $\rightarrow$ *Dict* $\rightarrow$ *Decl*$_1$-**set** *Dict*

Objective Construct the *Dict* descriptor of a service signal route

Parameters The service signal route containing

nm	The name of signal route
path	The first path
opath	The second optional path

Result An empty set of AS_1 definitions (because all the functions which transforms AS_1 definitions returns AS_1 definitions) and the *Dict* contribution containing the descriptor

Algorithm

Line 1	Decompose the first path in the service signal route.
Line 2	Construct the *Qual* which denotes the service signal route.
Line 3	Construct the *Quals* for the signals in the first path.
Line 4-5	Construct the *Qual* of the endpoints.
Line 6-16	If the second path is omitted then *osigset* is empty else *osigset* denotes the signals in the second path (the opposite directions as *sigset*).
Line 10	Decompose the second path.
Line 11-12	Construct the *Qual* of the endpoints in the second path
Line 12-13	The first endpoint of the first path must be equal to the second endpoint of the second path and the second endpoint of the first path must be equal to the first endpoint of the second path
Line 17	The same service must not be specified in both endpoints.
Line 18	Construct the service signal route descriptor

3.10 Creation of Implicit Channels and Signal Routes

This section contains the functions which for a block create:

- The implicit channels which are connected to the block if it contains exporting processes (for importing processes, it is the other block endpoint containing exporting processes which creates the channels)
- The implicit sub-channels which are connected to the block if it contains importing or exporting processes
- The connect definitions if the block contains a sub-structure which imports from- or exports to its environment
- The implicit signal routes and their associated connect definitions implied by import/export and implied by the valid input signal sets in the case where no signal routes are specified in AS_0
- The *ExpimpchanD* descriptors to be contained in the *BlockD* descriptor. These descriptors contains information about how the block should be interfaced such that implicit sub-channels can be created for the other block endpoint

The entry function which is applied in *transform-block* is *implicit-channels-and-signalroutes.*

implicit-channels-and-signal-routes(*decllist*)(*dict*) $\triangleq$ (3.10.1)

```
1  (let (exportset, importset) = imp-exp-in-processes(dict) in
2   let sigroutes = implicit-signalroutes(decllist)(dict),
3       blockimport = all-importeurs(dict),
4       blockexporteurs = all-exporteurs(dict),
5       (fatherimpm, fatherexpm) = father-block-import-export(dict) in
6   let exporttoenv = {closure | closure ∈ (exportset ∩ dom fatherexpm)},
7       importfromblock = {closure | closure ∈ (importset ∩ dom blockexporteurs)},
8       importfromenv = {closure | closure ∈ (importset ∩ dom fatherimpm)} in
9   let importblockmap = [closure ↦ blockexporteurs(closure) | closure ∈ importfromblock],
10      importenvmap = [cl ↦ ⟨(ENV, create-unique-name()) | 1 ≤ i ≤ len fatherimpm⟩ |
11                       cl ∈ importfromenv],
12      exportblockmap = [cl ↦ ⟨(bqual, create-unique-name()) | cl ∈ (exportset ∩ blockimport(bqual))⟩ |
13                        bqual ∈ dom blockimport],
14      exportenvmap = [cl ↦ ⟨(ENV, create-unique-name()) | 1 ≤ i ≤ len fatherexpm⟩ |
15                      cl ∈ exporttoenv] in
16  let importenvchannels = as0-channels(importenvmap, false)(dict),
17      exportblockchannels = as0-channels(exportblockmap, true)(dict),
18      exportenvchannels = as0-channels(exportenvmap, true)(dict) in
19  let connects = implicit-connects(importblockmap, exportblockmap)(dict) in
20  let decll = importenvchannels ⌢ exportblockchannels ⌢ exportenvchannels in
21  (decll, sigroutes, connects, importblockmap, exportblockmap))
```

type : $Decl_0^* \rightarrow Dict \rightarrow Decl_0^*\ Decl_0^*\ Decl_0^*\ ExpimpchanD\ ExpimpchanD$

Objective Construct the implicit channels, connect definitions and signal routes for a block

Parameters

decllist The definition list of the block. It is used for used for testing on whether there are explicit signal routes in the block

Result The channels to be defined at the same level as the block, the signal routes and the signal route connections to be defined in the block, the block sub-structure connections to be defined in the sub-structure (if any), the relation between the implicit channel names and the import signals it conveys (the channel name is deduced from the *ExpimpchanD* defined for the exporting block) and the relation between the implicit channel names and the export signals

Algorithm

Line 1	Let *exportset* denote the set of pairs of sort *Qual* and export name (*NameclosureD*s) for the processes in the block and let *importset* denote the *NameclosureD*s corresponding to export in the block.
Line 2	Let *sigroutes* denote the implicit signal routes created for the block.
Line 3	Let *blockimport* denote the relation between the block *Qual*s and the *NameclosureD*s for the imported names in the block.
Line 4	Let *blockexporteurs* denote the *ExpimpchanD* containing the exported names in all the other blocks and their names.
Line 5	Let *fatherimpm* denote the import *ExpimpchanD* descriptor for the block which contains this block and let *fatherexpm* denote the export *ExpimpchanD* descriptor for the block which contains this block
Line 6	The *NameclosureD*s for the names which are exported from this block to the environment of the scopeunit is the intersection between the *NameclosureD*s for the names exported from this block (*exportset*) and the *NameclosureD*s for the names also exported in the surrounding block
Line 7	The *NameclosureD*s for the names which are imported to this block from another block is the intersection between the *NameclosureD*s for the names imported from this block (*importset*) and the *NameclosureD*s for the names exported in another block
Line 8	The *NameclosureD*s for the names which are imported to this block from the environment of the scopeunit is the intersection between the *NameclosureD*s for the names imported from this block (*importset*) and the *NameclosureD*s for the names exported in the surrounding block.
Line 9	Construct the *ExpimpchanD* descriptor which contains information about the channel names conveying import signal names. The channel names are deduced from the descriptor contained in the *BlockD* for the exporting block.
Line 10-11	Construct the *ExpimpchanD* descriptor which contains information about the sub-channel names conveying import signal names to the environment of the block. In this case the channel names are created since the other endpoint is ENV.
Line 12-12	Construct the *ExpimpchanD* descriptor which contains information about the channel names conveying export signal names to other blocks (in the same scopeunit).
Line 14-15	Construct the *ExpimpchanD* descriptor which contains information about the sub-channel names conveying export signal names to the environment of the scopeunit.
Line 16	Construct the AS_0 sub-channels for the imported names in the block.
Line 17	Construct the AS_0 channels for the names which are exported to another block
Line 18	Construct the AS_0 channels for the names which are exported to the surrounding scopeunit.
Line 19	Construct the connect definitions for all the implicit sub-channels in the sub-structure of the block.
Line 20	Let *decll* denote the definitions which are to be defined at the same level as where the block is defined, that is, all the constructed channels.
Line 21	Return the channel definitions, the signal route definitions and the two *ExpimpchanD* maps

3.10.1 Creation of Implicit Channels

$as_0\text{-}channels(map, isexport)(dict) \triangleq$ (3.10.1.1)

```
1  (if map = [] then
2     ⟨⟩
3   else
4    (let clos ∈ dom map in
5     ⟨as0-channeldefs(map(clos), isexport, dict(clos))(dict)⟩ ↷
6     as0-channels(map \ {clos}, isexport)(dict)))
```

type : *ExpimpchanD Bool* → *Dict* → $Chandef_0^*$

Objective Construct the implicit channels associated to either import from the environment of a block or export to the environment of a block or export from a block to another block

Parameters

map	An *ExpimpchanD* containing information about the channel name and the name of one of the endpoint. The second endpoint is the current block, i.e. constructed from *dict*(SCOPEUNIT)
isexport	True if channels for export names in the current block are to be created

Result The constructed list of AS_0 channel definitions

Algorithm

Line 1	When through all the imported or exported names then return the empty list.
Line 4	Take the next *NameclosureD* for an imported or exported name.
Line 5	Construct the channel definition for that imported or exported name.
Line 6	Construct the channel definitions for the rest of the *NameclosureD*s

$as_0\text{-}channeldefs(list, isexport, signalnames)(dict) \triangleq$ (3.10.1.2)

```
1  (if list = ⟨⟩ then
2     ⟨⟩
3   else
4    (let level = dict(SCOPEUNIT),
5         blkid = as0-id(level),
6         mk-SignalnamesD(, xq, xr) = signalnames in
7     let signallist1 = ⟨mk-Id0(⟨⟩, xr)⟩,
8         signallist2 = ⟨mk-Id0(⟨⟩, xq)⟩,
9         (bqual, chan) = hd list in
10    let blkid' = as0-id(bqual) in
11    let (fromblock, toblock) = if isexport then (blkid, blkid') else (blkid', blkid) in
12    let direction1 = mk-Chanpath0(fromblock, toblock, signallist1),
13        direction2 = mk-Chanpath0(toblock, fromblock, signallist2) in
14    ⟨mk-Chandef0(chan, direction1, direction2, nil, chan)⟩ ↷
15    as0-channeldefs(tl list, isexport, signalnames)(dict)))
```

type : $(Otherend\ Channame_0)^*$ *Bool SignalnamesD* → *Dict* → $Chandef_0^*$

Objective For a given *SignalnamesD*, the two endpoint of channels and the channel names, the channel definitions conveying the xtREPLY signal and the xtQUERY signal are constructed

Parameters

list	A list of pairs each consisting of the other endpoint than the current one and the channel name. The list of pairs defines the channels to be created
isexport	**True** if the current block is the exporting block
signalnames	The *SignalnamesD* descriptor containing the xtREPLY and the xtQUERY signals corresponding to some *NameclosureD*

Result The list of AS_0 channel definitions

Algorithm

Line 1	When through, return the empty list (the function is recursive).
Line 4-14	Construct a channel definition for the next item in the list.
Line 4	Let *level* denote the current block *Qual.*
Line 5	Let *blkid* denote the identifier of the current block.
Line 6	Let *xq* and *xr* denote the xtQUERY signal and the xtREPLY signal to be conveyed by the channel.
Line 7-8	Construct the two signal lists which contain the signals.
Line 9	Let *bqual* denote the *Qual* of the other end block and let *chan* denote the channel name.
Line 10	Construct the identifier of the other end.
Line 11-11	If the current block is the exporter then *fromblock* denotes that block, otherwise *fromblock* denotes the other endpoint (in the case of a sub-channel).
Line 12-13	Construct the paths for the two signal lists.
Line 14	Construct the channel definition.
Line 15	Construct channel definitions for the rest of the list (*list*)

father-block-import-export(*dict*) $\triangleq$ (3.10.1.3)

```
1  (let level = dict(SCOPEUNIT) in
2   if len level > 2 then
3     (let fatherlevel = get-sur(get-sur(level)) in
4      let mk-BlockD(exp, imp, , ) = dict(fatherlevel) in
5      (exp, imp))
6    else
7     ([], []))
```

type : *Dict* → *ExpimpchanD ExpimpchanD*

Objective Extract the two *ExpimpchanD*s for the block which contains the current block

Result The *ExpimpchanD* for the export channels and the *ExpimpchanD* for the import channels

Algorithm

Line 1	Let *level* denote the *Qual* of the current block.
Line 2	If the block is contained in a sub-structure then
Line 3	Let *fatherlevel* denote the *Qual* of the block containing the sub-structure.
Line 4-5	Return the *ExpimpchanD*s for that block.
Line 8	If the current block is defined at the system level then the maps are empty as it is not possible to import/export from/to the environment of the system

imp-exp-in-processes(*dict*) $\triangleq$ (3.10.1.4)

```
1  (let level = dict(SCOPEUNIT) in
2   let pquals = {qual | qual ∈ dom dict ∧
3                        is-ProcessD(dict(qual)) ∧
4                        get-sur(qual) = level} in
5   if pquals = {} then
6     (let subblockquals = {qual | qual ∈ dom dict ∧
7                                  is-BlockD(dict(qual)) ∧
8                                  get-sur(get-sur(qual)) = level} in
9      let eiset = {imp-exp-in-processes(dict + [SCOPEUNIT ↦ qual]) | qual ∈ subblockquals} in
10     (union {imp | (imp, ) ∈ eiset}, union {exp | (, exp) ∈ eiset}))
11    else
12     (let eset = {vqual | vqual ∈ dom dict ∧
13                          is-VarD(dict(vqual)) ∧
14                          get-sur(vqual) ∈ pquals},
15          iset = {vqual | vqual ∈ dom dict ∧ is-ImportD(dict(vqual)) ∧
16                          get-sur(vqual) ∈ pquals} in
17      let expset =
18          {closure | (∃q ∈ eset)
19                     ((let mk-VarD(t, , exp, , ) = dict(q),
20                           (,vnm) = q[len q] in
21                       exp = EXPORTED ∧ closure = (t, vnm)))},
22          impset =
23          {closure | (∃q ∈ iset)
24                     ((let mk-ImportD(t) = dict(q),
25                           (,vnm) = q[len q] in
26                       closure = (t, vnm)))} in
27      (expset, impset)))
```

type : *Dict* → *NameclosureD*-**set** *NameclosureD*-**set**

Objective Extract the pairs of sort and variable names (*NameclosureD*s) for the export variables and for the import variables defined in the processes in a block- or - if the block contains no processes then for the processes in the contained sub-structure

Result The *NameclosureD*s for the exported variables and the *NameclosureD*s for the imported variables

Algorithm

Line 1 Let *level* denote the *Qual* of the current block.

Line 2-4 Let *pquals* denote the *Qual*s of the processes defined in the block.

Line 5 If there are no processes in the block then

Line 6-8 Let *subblockquals* denote the *Qual*s of the blocks contained in the sub-structure.

Line 9 Extract the set of pairs of *NameclosureD* for exported variables and for imported variables for the sub-blocks.

Line 10 Join all the export *NameclosureD*s in the set and join all the import *NameclosureD*s in the set.

Line 12-14 If the block contains processes then let *eset* denote the set of variable *Qual*s for the variables defined in the processes.

Line 15-16 Let *iset* denote the set of variable *Qual*s for the import variables defined in the processes.

Line 17-21 Let *expset* denote the *NameclosureD*s for the exported variables, i.e. the set of pairs of sort (*t*) and exported (*exp* = EXPORTED) variable name (*vnm*).

Line 22-26 Let *impset* denote the *NameclosureD*s for the imported variables.

Line 27 Return the two sets

all-importeurs(*dict*) $\triangleq$ (3.10.1.5)

```
1  (let level = dict(SCOPEUNIT) in
2   let sur = get-sur(level) in
3   let blockqset = {qual | qual ∈ dom dict ∧
4                           is-BlockD(dict(qual)) ∧
5                           get-sur(qual) = sur} \ {level} in
6   let expimpmap = [q ↦ imp-exp-in-processes(dict + [SCOPEUNIT ↦ q]) | q ∈ blockqset] in
7   [q ↦ closset | q ∈ dom expimpmap ∧ (, closset) = expimpmap(q)])
```

type: *Dict* → (*Blockqual* $\overset{m}{\to}$*NameclosureD*-**set**)

Objective Extract information about the relation between the *NameclosureD*s for import and the importing block. The current block is excluded in the relation (map)

Algorithm

Line 1	Let *level* denote the current block.
Line 2	Let *sur* denote the *Qual* of the surrounding block sub-structure.
Line 3-5	Construct a set consisting of all blocks in the sub-structure except the current one.
Line 6	Construct a map from importing blocks to the pairs of *NameclosureD*s sets for the block.
Line 7	Return a map from importing blocks to the *NameclosureD* for the imported variables in the blocks

all-exporteurs(*dict*) $\triangleq$ (3.10.1.6)

```
1  (let level = dict(SCOPEUNIT) in
2   let sur = get-sur(level) in
3   let blockqset = {qual | qual ∈ dom dict ∧
4                           is-BlockD(dict(qual)) ∧
5                           get-sur(qual) = sur} \ {level} in
6   get-exporteurs(blockqset)(dict))
```

type: *Dict* → *ExpimpchanD*

Objective Construct the (composite) *ExpimpchanD* which includes the exported variables in all the other blocks (except the current one) which exports.

Algorithm

Line 1	Let *level* denote the *Qual* of the current block.
Line 2	Let *sur* denote the *Qual* of the surrounding block sub-structure.
Line 3-5	Let *blockqset* denote the set of *Qual*s for all the other blocks in the sub-structure.
Line 6	Traverse recursively through the set of *Qual*s while joining the *ExpimpchanD*s for the blocks

get-exporteurs(*blockqset*)(*dict*) $\triangleq$ (3.10.1.7)

```
1  (if blockqset = {} then
2     []
3   else
4    (let q ∈ blockqset in
5     let blkqual = dict(SCOPEUNIT) in
6     let mk-BlockD(expm,,,) = dict(q) in
7     let expm' = [closure ↦ tup | closure ∈ dom expm ∧
8                                  (let tup' = expm(closure) in
9                                   if (∃i ∈ ind tup')(s-Otherend(tup[i]) = blkqual) then
10                                    (let i ∈ ind tup' be s.t. s-Otherend(tup[i]) = blkqual in
11                                     tup = ⟨tup[i]⟩)
12                                   else
13                                    false)] in
14    let rest = get-exporteurs(blockqset \ {q})(dict) in
15    [clos ↦ (rest + expm')(clos) | clos ∈ (dom rest − dom expm')] +
16    [clos ↦ newlist | clos ∈ (dom rest ∩ dom expm') ∧ newlist = rest(clos) ⌢ expm'(clos)]))
```

type : *Blockqual*-**set** → *Dict* → *ExpimpchanD*

Objective Construct the composite *ExpimpchanD* which includes the information about export in all blocks

Parameters

blockqset — The set of (remaining) blocks to be dealt with (the function is recursive)

Result The constructed composite *ExpimpchanD*

Algorithm

Line 1 — When having dealt with all the blocks, return the empty map.

Line 4 — Let *q* denote the next block *Qual* in the set.

Line 5 — Let *blkqual* denote the *Qual* of the current block.

Line 6 — Let *expm* denote the *ExpimpchanD* corresponding to export from the block.

Line 7-13 — Restrict the *ExpimpchanD* map such that it only includes the elements where the other endpoint is the current block.

Line 14 — Construct the composite *ExpimpchanD* for the rest of the blocks.

Line 15-16 — Return the *ExpimpchanD* where the lists in the two maps (*expm'* and *rest*) have been concatenated. Line 15 defines the unchanged entries and line 16 defines the entries which are constructed by concatenating the lists for two entries from the maps

3.10.2 Creation of Implicit Signalroutes

implicit-signalroutes(*decll*)(*dict*) $\triangleq$ (3.10.2.1)

```
1  (let existexplicit = (∃d ∈ decll)(is-Sigroutedef0(d)) in
2   if ¬existexplicit ∧ (∃mk-Prdef0(, inst, , , , , ) ∈ elems decll)(inst = nil) then
3     exit("§2.5.2: Valid input signal set must be specified when no signal routes are specified")
4    else
5     (let bqual = dict(SCOPEUNIT) in
6      let cqualset =
7          {cqual ∈ dom dict | is-ChannelD(dict(cqual)) ∧
8                              (let mk-ChannelD(endp1, endp2, , ) = dict(cqual) in
9                               bqual ∈ {endp1, endp2})} in
10     let pathset = {p ∈ dom dict | is-ProcessD(dict(p)) ∧ is-local(p)(dict)} in
11     let pathpairset ∈ (Processqual-set)-set be s.t. (∀e ∈ pathpairset)(card e = 2) ∧
12         union pathpairset = pathset ∧
13         (∀p1, p2 ∈ pathset)(p1 ≠ p2 ⊃ {p1, p2} ∈ pathpairset) in
14     make-as0-local-signalroutes(pathpairset, existexplicit)(dict) ⌢
15     make-as0-env-signalroutes(cqualset, existexplicit)(dict)))
```

type: $Decl_0^* \rightarrow Dict \rightarrow Decl_0^*$

Objective Construct the AS_0 implicit signal routes for a block

Algorithm

Line 1	Let *existexplicit* be **true** if there are specified any signal routes in the block.
Line 2-3	If no explicit signal routes are specified for the block then all contained process definitions must contain a valid input signal set.
Line 5	Extract the *Qual* of the block.
Line 6-9	Extract the descriptors for those channels which have the block as one of the endpoints.
Line 10	Construct a set containing the *Qual* of all the processes in the block.
Line 11-13	Construct a set of *Processqual* sets where each contained set contains two elements (line 11), the sets contains only those process *Qual*s which are defined in the block (line 12) and the set contains all possible combinations of sets (line 13).
Line 14	Construct the signal routes between processes in the block and
Line 15	Construct the signal routes connected to the environment of the block

make-as$_0$-local-signalroutes(*pathset*, *existexplicit*)(*dict*) $\triangleq$ (3.10.2.2)

```
1  (if pathset = {} then
2     ⟨⟩
3   else
4    (let {p1, p2} ∈ pathset in
5     let mk-ProcessD(, insig1, outsig1, ) = dict(p1),
6         mk-ProcessD(, insig2, outsig2, ) = dict(p2) in
7     let (insig1', outsig1') = if existexplicit then
8                                 import-export-signals(dict)
9                               else
10                                (insig1, outsig1),
11        (insig2', outsig2') = if existexplicit then
12                                import-export-signals(dict)
13                              else
14                                (insig2, outsig2) in
15    let as0p1id = as0-id(p1),
16        as0p2id = as0-id(p2) in
17    let (com1sig, com2sig) = (insig1' ∩ outsig2', insig2' ∩ outsig1') in
18    let as0sigset1 = {as0-id(id) | id ∈ com1sig},
19        as0sigset2 = {as0-id(id) | id ∈ com2sig} in
20    let com1 = if com1sig = {}
21                 then nil
22                 else mk-Sigroutepath0(as0p1id, as0p2id, as0sigset1),
23        com2 = if com2sig = {}
24                 then nil
25                 else mk-Sigroutepath0(as0p2id, as0p1id, as0sigset2) in
26    let (com', com'') = if com1 = nil then (com2, com1) else (com1, com2) in
27    let route0 =
28        if com' = nil ∧ com'' = nil
29          then ⟨⟩
30          else (let nm0 = create-unique-name() in
31                ⟨mk-Sigroutepath0(nm0, com', com'')⟩) in
32    route0 ⌢ make-as0-local-signalroutes(pathset \ {{p1, p2}}, existexplicit)(dict)))
```

type : (*Processqual*-**set**)-**set** *Bool* → *Dict* → *Sigroutedef*$_0^*$

Objective Construct AS$_0$ signal routes which leads between processes in a block

Parameters

pathset	The set of pairs (sets containing two elements) of process
existexplicit	True if there are specified signal routes for the block in AS$_0$

Result A list of AS$_0$ signal routes

Algorithm

Line 1	When through, return the empty list (the function is recursive).
Line 4	Let *p1* and *p2* denote a combination of processes to be dealt with.
Line 5-6	Decompose the descriptors of the processes.
Line 7-11	If signal routes are specified for the process in AS$_0$ then signal routes are only constructed for the import/export signals.
Line 15-16	Make the AS$_0$ identifiers for the processes.
Line 17	Let *com1sig* denote the signals which leads from *p2* to *p1* and let *com2sig* denote the signals which leads from *p1* to *p2*.
Line 18	Construct the AS$_0$ identifier set corresponding to *com1sig*.
Line 19	Construct the AS$_0$ identifier set corresponding to *com2sig*.
Line 20-25	Let *com1* and *com2* denote the AS$_0$ communication paths from *p2* to *p1* respectively from *p1* to *p2* (**nil** if a path contains no signals).

Line 26-31	Let $route_0$ denote a signal route which contains the signal between the two processes.
Line 32	Join the signal route definition with those for the rest of pairs of processes

make-as$_0$*-env-signalroutes*(*cqualset*, *existexplicit*)(*dict*) $\triangleq$ (3.10.2.3)

```
1  (if cqualset = {} then
2     ⟨⟩
3   else
4    (let bqual = dict(SCOPEUNIT) in
5     let cqual ∈ cqualset in
6     let pqualset = {pqual | pqual ∈ dom dict ∧ is-local(pqual)(dict)} in
7     let sigroutes = make-as0-channel-signalroutes(cqual, pqualset, existexplicit)(dict) in
8     if sigroutes = {} then
9       make-as0-env-signalroutes(cqualset \ {cqual}, existexplicit)(dict)
10      else
11       (let as0chanid = as0-id(cqual),
12            routenameset = {nm | (∃mk-Sigroutedef0(nm',,) ∈ sigroutes)(nm' = nm)} in
13        let as0list = ⟨as0-id(bqual ⌢ ⟨(SIGNALROUTE, nm)⟩) | nm ∈ routenameset⟩ in
14        ⟨sigroute | sigroute ∈ sigroutes⟩ ⌢
15        ⟨mk-Connect0(as0chanid, as0list)⟩ ⌢
16        make-as0-env-signalroutes(cqualset \ {cqual}, existexplicit)(dict))))
```

type: *Channelqual-set Bool* → *Dict* → *Decl*$_0^*$

Objective Construct the implicit AS_0 signal routes which are connected to the environment of the block

Parameters

cqualset	The set of channels to which the block is connected
existexplicit	**True** if signal routes are specified in the concrete syntax for the block

Result The constructed list of AS_0 signal routes and the appropriate connect definitions

Algorithm

Line 1	When through, return the empty list (the function is recursive).
Line 4	Let *bqual* denote the *Qual* of the block.
Line 5	Take a next channel *Qual* to be dealt with.
Line 6	Let *pqualset* denote the *Qual*s of the processes in the block.
Line 7	Construct the signal routes connected to the channel.
Line 8	If there are no signal routes connected to the channel (this is an error, but it is checked when the signal routes are transformed into AS_1) or signal routes already exist for the channel then continue to consider the rest of the channels.
Line 11	Construct the AS_0 identifier of the channel.
Line 12	Extract the names of the constructed signal routes.
Line 13	Turn the names into identifiers.
Line 14	Return the signal routes and
Line 15	Their associated connect definition and
Line 16	The signal routes and connects corresponding to the rest of the channels

make-as$_0$-channel-signalroutes(*cqual*, *pqualset*, *existexplicit*)(*dict*) $\triangleq$ (3.10.2.4)

```
1  (if pqualset = {} then
2     ⟨⟩
3   else
4    (let mk-ChannelD(end1, , sigset1, sigset2, ) = dict(cqual) in
5     let pqual ∈ pqualset in
6     let mk-ProcessD(, insig, outsig, ) = dict(pqual) in
7     let (insig', outsig') = if existexplicit then
8                               import-export-signals(dict)
9                             else
10                            (insig, outsig) in
11    let bqual = dict(SCOPEUNIT) in
12    let (fromprocesssignals, toprocesssignals) = if bqual = end1 then
13                                                   (sigset1 ∩ outsig', sigset2 ∩ insig')
14                                                 else
15                                                  (sigset2 ∩ outsig', sigset1 ∩ insig') in
16    if fromprocesssignals ∪ toprocesssignals = {} then
17      make-as0-channel-signalroutes(cqual, pqualset \ {pqual}, existexplicit)(dict)
18     else
19      (let routenm = create-unique-name() in
20       let as0pid = as0-id(pqual) in
21       let as0fromsignals = {as0-id(id) | id ∈ fromprocesssignals},
22           as0tosignals = {as0-id(id) | id ∈ toprocesssignals} in
23       let path1 = if fromprocesssignals = {}
24                   then nil
25                   else mk-Sigroutepath0(as0pid, ENV, as0fromsignals),
26           path2 = if toprocesssignals = {}
27                   then nil
28                   else mk-Sigroutepath0(ENV, as0pid, as0tosignals) in
29       let (path', path'') = if path1 = nil then (path2, path1) else (path1, path2) in
30       ⟨mk-Sigroutedef0(routenm, path', path'')⟩ ⌢
31       make-as0-channel-signalroutes(cqual, pqualset \ {pqual}, existexplicit)(dict))))
```

type : *Channelqual Processqual*-set *Bool* → *Dict* → *Sigroutedef*$_0^*$

Objective Construct the implicit signal routes corresponding to a given channel

Parameters

cqual	The *Qual* of the channel
pqualset	The *Quals* of the processes in the block
existexplicit	**True** if signal routes are specified in the block

Result The constructed list of AS$_0$ signal routes

Algorithm

Line 1	When through, return the empty list (the function is recursive).
Line 4	Let *end1* denote the endpoint of the channel from which the signals *sigset1* can be sent and from which the signals *sigset2* can be received.
Line 5	Let *pqual* denote the next process to be considered.
Line 6	Let *insig* denote the input signals of the process and let *outsig* denote the output signals of the process.
Line 7	If signal routes are specified in the concrete syntax then the implicit signal routes are only those implied from import/export.
Line 11	Let *bqual* denote the *Qual* of the surrounding block.
Line 12-15	Let *fromprocesssignals* denote the signals which are sent from the process via the channel and let *toprocesssignals* denote the signals which are received through the channel.

Line 16-17	If there are no signals sent or received through the channel then continue to consider the rest of the processes.
Line 19	Create an implicit name for the signal route to be constructed.
Line 20	Construct the AS_0 identifier of the process.
Line 21	Turn the output signals into AS_0 signal identifiers.
Line 22	Turn the input signals into AS_0 signal identifiers.
Line 23-25	If there are any signals sent from the process through the channel then construct the appropriate $Sigroutepath_0$.
Line 26-28	If there are any signals received by the process through the channel then construct the appropriate $Sigroutepath_0$.
Line 29	Let *path'* denote the path which contains signals (the first path) and let *path''* denote the second optional path.
Line 30	Construct and return a signal route connected to the channel and construct signal routes for the rest of the processes connected to the channel

3.10.3 Creation of Implicit Connect Statements

implicit-connects(*exp*, *imp*)(*dict*) $\triangleq$ (3.10.3.1)

```
1  (let level = dict(SCOPEUNIT) in
2   let expset =
3       {expmap | (∃qual ∈ dom dict)
4                 (is-BlockD(dict(qual)) ∧
5                  get-sur(qual) = level ∧
6                  dict(qual) = mk-BlockD(expmap,,,))} in
7   let impset =
8       {impmap | (∃qual ∈ dom dict)
9                 (is-BlockD(dict(qual)) ∧
10                 get-sur(qual) = level ∧
11                 dict(qual) = mk-BlockD(,impmap,,))} in
12  let outconnect =
13      if exp = [] then
14        ⟨⟩
15       else
16        (let closure ∈ dom exp in
17         gen-connect(expset, closure, exp(closure), exp \ {closure})),
18      inconnect =
19      if imp = [] then
20        ⟨⟩
21       else
22        (let closure ∈ dom imp in
23         gen-connect(impset, closure, imp(closure), imp \ {closure})) in
24  outconnect ↷ inconnect)
```

type: *ExpimpchanD ExpimpchanD* → *Dict* → $Connect_0^*$

Objective Construct the AS_0 channel connections for the implicit channels implied by import/export

Parameters

exp	The *ExpimpchanD* export map for the block containing the blocks to which the implicit sub-channels are associated
imp	The *ExpimpchanD* import map for the block containing the blocks to which the implicit sub-channels are associated

Result The list of AS_0 channel connections

Algorithm

Line 1	Let *level* denote the *Qual* of the current block sub-structure.
Line 2-6	Construct a set of export *ExpimpchanD* maps. There is one map for each sub-block in the sub-structure.
Line 7-11	Construct a set of import *ExpimpchanD* map. There is one map for each sub-block in the sub-structure.
Line 12-17	Construct the channel connections for the channels implied from export from the sub-structure. *closure* denotes the first *NameclosureD* to be treated.
Line 18-23	Construct the channel connections for the channels implied from import to the sub-structure. *closure* denotes the first *NameclosureD* to be treated

gen-connect(*mapset*, *closure*, *list*, *map*) $\triangleq$ (3.10.3.2)

```
1  (if list = ⟨⟩ then
2     if map = [] then
3        ⟨⟩
4      else
5       (let closure' ∈ dom map in
6         gen-connect(mapset, closure', map(closure'), map \ {closure'}))
7    else
8     (let (mapset', chanlist) = gen-connect-elem(mapset, closure, ⟨⟩) in
9      let (,channame) = hd list in
10     let connect = mk-Connect0(channame, chanlist) in
11     ⟨connect⟩ ⌢ gen-connect(mapset', closure, tl list, map)))
```

type : *ExpimpchanD*-**set** *NameclosureD* (*Otherend Channame*$_0$)* *ExpimpchanD* $\rightarrow$ *Connect*$_0{}^+$

Objective Construct the channel connections implied from import or export of a variable to/from a block sub-structure

Parameters

expimpset	The set of export *ExpimpchanD*s or import *ExpimpchanD*s for the blocks in the block sub-structure, i.e. information about export or import in the sub-blocks
closure	The *NameclosureD* to be treated
list	Contains the list of implicit channels connected to the sub-structure which conveys the implicit signals corresponding to the *NameclosureD* (*closure*)
map	The *ExpimpchanD* for the rest of the *nameclosureD*s to be treated for the sub-structure

Result A list of AS_0 channel connections

Algorithm

Line 1	When through the channels connected to the block sub-structure which correspond to *closure* then
Line 2	If there are no more *NameclosureD*s to be treated then return else
Line 5	Let *closure* denote the next *NameclosureD* to be treated.
Line 6	Create the channel connections for the implied sub-channels corresponding to *closure* and create the channel connections for the rest of the *NameclosureD*s.
Line 8	Update the set of *ExpimpchanD* maps (*mapset'*) for the sub-blocks such that the entries corresponding to *closure* which leads to the environment of the sub-structure are excluded. Also construct the AS_0 sub-channel identifier list to be used in the channel connection.

Line 9	Let *channame* denote the name of the implicit channel connected to the sub-structure.
Line 10	Construct the channel connection.
Line 11	Join the channel connection with the channel connections for the rest of the implicit sub-channels

gen-connect-elem(*mapset*, *closure*, *chanlist*) $\triangleq$ (3.10.3.3)

```
 1  (if mapset = {} then
 2     ({},chanlist)
 3    else
 4     (let map ∈ mapset in
 5      let (mapset', chanlist') = gen-connect-elem(mapset \ {map}, closure, chanlist) in
 6      let clist = map(closure) in
 7      if (∃index ∈ ind clist)(s-Otherend(clist[index]) = ENV) then
 8        (let index be s.t. s-Otherend(clist[index]) = ENV in
 9         let clist' = ⟨clist[i] | 1 ≤ i ≤ len clist ∧ i ≠ index⟩ in
10         let map' = map + [closure ↦ clist'] in
11         let (,chan) = clist[index] in
12         (mapset' ∪ {map'}, chanlist' ⌢ ⟨mk-Id0(⟨⟩, chan)⟩))
13       else
14        (mapset', chanlist')))
```

type: *ExpimpchanD*-**set** *NameclosureD* Id_0^* $\rightarrow$ *ExpimpchanD*-**set** Id_0^*

Objective Extract the channel identifiers to be used in a channel connection which is implied by a certain import or export variable. The channel identifiers are found in the *ExpimpchanD* descriptors of the various sub-blocks. After having extracted a channel identifier, the information is removed from the *ExpimpchanD* descriptor. The function also returns the modified *ExpimpchanD* descriptors

Parameters

mapset	The set of *ExpimpchanD* maps. There is one map for each sub-block
closure	The *NameclosureD* to which the implicit sub-channels are attached
chanlist	The list of channel identifiers which is created recursively

Result The modified set of *ExpimpchanD* maps and the constructed sub-channel identifier list

Algorithm

Line 1	When through all the *ExpimpchanD* descriptors then return the constructed channel identifier list.
Line 4	Let *map* denote a next *ExpimpchanD* descriptor to be considered.
Line 5	Modify the set of *ExpimpchanD* maps corresponding to the rest of the sub-blocks and construct the appropriate sub-channels leading to the rest of the sub-blocks
Line 6	*clist* contains information about all the sub-channel connected to a sub-block conveying implicit signals corresponding to the *NameclosureD* (*closure*)
Line 7	If one of the sub-channels are leading to the environment of the sub-structure then
Line 8	Let *index* be an index to the element containing the name of the channel leading to the environment.
Line 9	Exclude that element from the list.
Line 10	Modify the *ExpimpchanD* map with the new list.
Line 11	Let *chan* denote the sub-channel name.

Line 12	Return the modified *ExpimpchanD* map set and the channel identifier list.
Line 14	If there is no sub-channel leading to the environment of the substructure then return the *ExpimpchanD* map and the channel identifier list unchanged

3.11 Utility Functions

get-predef-sort(*quot*)(*dict*) $\triangleq$ (3.11.1)

```
1  (let nm = mk-Name0(quot, nil),
2       level = dict(SCOPEUNIT) in
3   let sys = level[1] in
4   sys ⌢ ⟨(TYPE, nm)⟩)
```

type : *Char** $\rightarrow$ *Dict* $\rightarrow$ *Sortqual*

Objective Extract the *Qual* of a predefined sort

Parameters

quot	The sequence of characters representing the spelling of the sort name

Algorithm

Line 1	Construct an AS_0 name from the spelling.
Line 2-3	Let *sys* denote the *Qual* of the system level.
Line 4	Return the *Qual* for the predefined sort (defined at system level)

get-parent(*qual*)(*dict*) $\triangleq$ (3.11.2)

```
1  (if qual ∈ dom dict then
2     (if is-SortD(dict(qual)) then
3        qual
4        else
5        (let mk-SyntypeD(q,,,) = dict(qual) in
6         get-parent(q)(dict \ {qual})))
7    else
8     exit("§5.4.1.9: Syntype is defined in terms of itself"))
```

type : *Sortqual* $\rightarrow$ *Dict* $\rightarrow$ *Sortqual*

Objective From a *Qual* denoting either a sort or a syntype, extract the parent sort (in the case of a syntype)

Algorithm

Line 1	If the *Qual* is in the *Dict* then
Line 2-3	If the *Qual* denotes a sort then return the *Qual* else
Line 5-6	If the *Qual* denotes a syntype then extract the parent sort, but exclude the syntype *Qual* from *Dict* such that it can be detected if the syntype is recursively defined

is-local(*qual*)(*dict*) $\triangleq$ (3.11.3)

```
1  qual = ENV ∨
2  dict(SCOPEUNIT) = get-sur(qual)
```

type : (*Qual* | ENV) $\rightarrow$ *Dict* $\rightarrow$ *Bool*

Objective Return true if a *Qual* is defined local to a scopeunit

Algorithm

Line 1 If *qual* is the quotation ENV (used in connection with channels and signal routes) or

Line 2 They are defined in the same scopeunit then return true

process-level(*qual*) $\triangleq$ (3.11.4)

```
1  (let (q, ) = qual[len qual] in
2   if q ∈ {PROCEDURE, SERVICE} then
3     process-level(get-sur(qual))
4    else
5     qual)
```

type: *Qual* → *Qual*

Objective From a *Qual* denoting a process, procedure or service extract the *Qual* of the surrounding process

Algorithm

Line 1-5 If the *Qual* denotes a procedure or a service then apply the function on the *Qual* of the surrounding scopeunit

process-or-service-level(*qual*) $\triangleq$ (3.11.5)

```
1  (let (q, ) = qual[len qual] in
2   if q = PROCEDURE then
3     process-or-service-level(get-sur(qual))
4    else
5     qual)
```

type: *Qual* → *Qual*

Objective From a *Qual* denoting a process, procedure or service extract the *Qual* of the surrounding process or service

Algorithm

Line 1-5 If the *Qual* denotes a procedure then apply the function on the *Qual* of the surrounding scopeunit

get-sur(*qual*) $\triangleq$ (3.11.6)

```
1  ⟨qual[i] | 1 ≤ i ≤ len qual − 1⟩
```

type: *Qual* → *Qual*

Objective From a *Qual*, return the *Qual* of the surrounding scopeunit by removing the last element.

get-inherited-parent(*qual*)(*dict*) $\triangleq$ (3.11.7)

```
1  (if qual ∈ dom dict then
2      cases dict(qual):
3       (mk-SortD(, pqual, , )
4           → if pqual = nil then qual else get-inherited-parent(pqual)(dict \ {qual}),
5        mk-SyntypeD(pqual, , , )
6           → get-inherited-parent(pqual)(dict \ {qual}))
7    else
8     exit("§5.4.1.9: Syntype is defined in terms of itself"))
```

type : *Sortqual* → *Dict* → *Sortqual*

Objective From a *Qual* denoting either a sort or a syntype, extract the parent sort for an inheriting sort.

Algorithm

Line 1 If the *Qual* is in the *Dict* then

Line 3 If the *Qual* denotes a none-inheriting sort then return the *Qual* else if the *Qual* denotes an inheriting sort then extract the parent sort, but exclude the sort *Qual* from *Dict* such that it can be detected if the sort inherits from itself.

Line 5 If the *Qual* denotes a syntype then extract the parent sort, but exclude the syntype *Qual* from *Dict* such that it can be detected if the sort inherits from itself.

3.12 Generation of Auxiliary Information

This section contains the functions which generates the *Auxiliary* information to be used by the underlying system. The entry function is *generate-auxiliary-information*

generate-auxiliary-information(*extparms*)(*dict*) $\triangleq$ (3.12.1)

```
1  (let (starttime, timeunit) = derive-time-inf(extparms)(dict) in
2   let tick = make-tick-function(timeunit)(dict) in
3   let terminf = make-term-inf(dict) in
4   let expired = make-expired-function(dict) in
5   let delayf = derive-delay-inf(extparms)(dict) in
6   ((tick, starttime), terminf, expired, delayf))
```

type : *External-Information* → *Dict* → *Auxiliary-Information*

Objective Derive the auxiliary information to be used by the underlying system (annex F.3)

Parameters

extparms The external information from which the required objects are extracted

Result The *Auxiliary-Information*

Algorithm

Line 1 Derive the TIME literal denoting the starttime and the DURATION literal denoting the atomic time unit

Line 2 Construct the function which is used for updating the current time (i.e. it adds *timeunit* to the current time)

Line 3 Construct the identifier for the PID sort and for the three literals NULL, TRUE and FALSE

Line 4	Construct the function which is used for testing whether a given timer has expired (i.e. it compares two TIME values)
Line 5	Derive the function which is used in the model of indeterministic delay in channels
Line 6	Construct and return the *Auxiliary-Information*

make-tick-function(*timeunit*)(*dict*) $\triangleq$ (3.12.2)

```
1  (let durationq = get-predef-sort("DURATION")(dict),
2       timeq = get-predef-sort("TIME")(dict) in
3   let mk-OperatorD(,, nq,) = dict(timeq ⌢ ⟨(OPERATOR, (PLUS, ⟨timeq, durationq⟩, timeq))⟩) in
4   let as1 id = make-as1-identifier(nq)(dict) in
5   let f(value) = mk-Ground-term1((as1 id, ⟨value, timeunit⟩)) in
6   f)
```

type: *Literal-operator-identifier*$_1$ $\rightarrow$ *Dict* $\rightarrow$ (*Ground-term*$_1$ $\rightarrow$ *Ground-term*$_1$)

Objective Construct a function which updates the time with a given atomic time-unit. The function is used in the abstract SDL machine. The current value of NOW is applied and the new value is returned

Parameters

timeunit	The timeunit represented by an AS_1 literal.

Algorithm

Line 1	Construct the *Qual* of the DURATION sort.
Line 2	Construct the *Qual* of the TIME sort.
Line 3	Decompose the descriptor of the "+" operator defined for the TIME sort. The *Qual* of the operator is formed by concatenating the TIME *Qual* with the element containing the entity class (OPERATOR), the name of the operator (PLUS), the argument sorts (*timeq* and *durationq*) and the result sort (*timeq*). Refer to the definition of *Operatorqualelem*.
Line 4	Construct the AS_1 identifier of the operator.
Line 5-6	Construct and return a Meta-IV function which when given an SDL value, adds the timeunit to the value

make-term-inf(*dict*) $\triangleq$ (3.12.3)

```
1  (let bqual = get-predef-sort("BOOLEAN")(dict) in
2   let pidqual = get-predef-sort("PID")(dict) in
3   let pid1 = make-as1-identifier(pidqual)(dict),
4       null1 = make-as1-identifier(pidqual ⌢ ⟨(LITERAL, mk-Name0("NULL", nil))⟩)(dict),
5       true1 = make-as1-identifier(bqual ⌢ ⟨(LITERAL, mk-Name0("TRUE", nil))⟩)(dict),
6       false1 = make-as1-identifier(bqual ⌢ ⟨(LITERAL, mk-Name0("FALSE", nil))⟩)(dict) in
7   (pid1, null1, true1, false1))
```

type: *Dict* $\rightarrow$ *Term-Information*

Objective Construct the *Term-Information* to be used in the Dynamic Semantics (see the domain definition of *Term-Information*)

Result The *Term-Information* consisting of the identifiers for the PID sort, the NULL literal, the TRUE literal and the FALSE literal

make-expired-function(*dict*) $\triangleq$ (3.12.4)

```
1  (let bqual = get-predef-sort("BOOLEAN")(dict) in
2   let geop = mk-Qualop0(bqual, mk-Quotedop0(GE)) in
3   let expiredf(currenttime, expiredtime) =
4       eval-simple-expr(mk-Operatorapp0(geop, ⟨currenttime, expiredtime⟩), "BOOLEAN")(dict) in
5   expiredf)
```

type: *Dict* → *Is-expiredF*

Objective Construct a function which tests on whether a given time value exceeds an expiration time value, i.e. make the function which returns the Meta-IV value **true** if

`currenttime >= expiredtime`

where `currenttime` and `expiredtime` are formal parameters.

The function is to be used in the Dynamic Semantics

Algorithm

Line 1	Construct the *Qual* of the BOOLEAN sort.
Line 2	Construct the AS_0 identifier of the ">=" operator
Line 3-5	Construct and return the *Is-Expired* function (*expiredf*)

3.13 Global Constant Mappings

In the following, two global mappings *name-to-name*$_1$ and *exportmap* are defined.

3.13.1 Relation between AS_0 names and AS_1 names

Names in AS_1 does not have the same representation as in AS_0, since the spelling etc. of names is of no concern for the interpretation. Therefore, AS_0 names (*Name*$_0$), character strings (*String*$_0$) and Quoted operators (*Quotedop*$_0$) are transformed into AS_1 names (*Name*$_1$) by means of a global mappings (*name-to-name*$_1$).

let *name-to-name*$_1$ **be s.t.** $(\forall nm_1 \in \textbf{rng}\ name\text{-}to\text{-}name_1)(\textbf{is-}Name_1(nm_1)) \wedge$
$(\forall(nm1, nm2) \in (Name_0 \mid String_0 \mid Quotedop_0))$
$((\{nm1, nm2\} \subset \textbf{dom}\ name\text{-}to\text{-}name_1) \wedge$
$(nm1 \neq nm2 \supset name\text{-}to\text{-}name_1(nm1) \neq name\text{-}to\text{-}name_1(nm2)))$ **in**

Objective Construct a map which converts AS_0 names, strings and quoted operators to AS_1 names.

Algorithm Construct the map that every possible AS_0 name, string or quoted operator is in the domain of the map and every two entities maps onto different AS_1 names.

3.13.2 Relation between import/export names and implicit signal names

In SDL, some implicit signals are attached to each export variable. Two export variables have the same implicit signals attached if they have the same name (spelling) and they are of the same data sort. In order to manage this "sharing" of signals, a special constant mapping *exportmap* is used, which maps pairs of export variable sort and export variable name into a descriptor of the attached signal names:

let *exportmap* = [*closure* $\mapsto$ **mk**-*SignalnamesD*(*create-unique-name*(),
create-unique-name(),
create-unique-name()) | *closure* $\in$ *NameclosureD*] **in**

The domain of *exportmap* is defined in the following:

1	*Exportmap*	=	*NameclosureD* $\overrightarrow{m}$ *SignalnamesD*
2	*NameclosureD*	=	*Sortqual* $Name_0$
3	*SignalnamesD*	::	*Impliednm Xquerynm Xreplynm*
4	*Impliednm*	=	$Name_0$
5	*Xquerynm*	=	$Name_0$
6	*Xreplynm*	=	$Name_0$

The signal descriptor (*SignalnamesD*) consists of

Impliednm	The implicit variable attached to each export variable.
Xquerynm	The xtQUERY signal send by an importer.
Xreplynm	The xtREPLY signal send by the exporter and carrying a value of sort *Sortqual*.

3.14 Informal Functions

apply-generic-parameters(*genericsystem*, *extparameters*) $\triangleq$ (3.14.1)

1 /* *Turn a generic system definition into a concrete system definition* */

type: Sys_0 *External-Information* $\rightarrow$ Sys_0

SDL does not define how generic parameters are applied. This informal function takes an AS_0 system definition and the external information which includes some actual generic parameters and returns an AS_0 system definition containing no generic parameters, that is, a system definition which contains no external synonyms and no informal text in the answers of the option actions

derive-time-inf(*extparms*)(*dict*) $\triangleq$ (3.14.2)

1 /* *Derive the start time and the unit for time from the external information* */

type: *External-Information* $\rightarrow$ *Dict* $\rightarrow$ *Literal-operator-identifier*$_1$ *Literal-operator-identifier*$_1$

This informal function returns the *Literal-operator-identifier*$_1$ of the sort TIME denoting the starting time of the system and the *Literal-operator-identifier*$_1$ of the sort DURATION denoting the increment value of the absolute time (both used in the model for the NOW expression, see Annex F.3)

derive-delay-inf(*extparms*)(*dict*) $\triangleq$ (3.14.3)

1 /* *Derive the delay function from the external information* */

type: *External-Information* $\rightarrow$ *Dict* $\rightarrow$ *DelayF*

This informal function derives from the external information the imperative function (*DelayF*) used in Annex F.3 for modelling the indeterministic delay in channels.

eval-expr(*expr*, *sort*)(*dict*) $\triangleq$ (3.14.4)

1 /* *Evaluate the expression* */

type: *Ground-expression*$_1$ $Char^+$ $\rightarrow$ *Dict* $\rightarrow$ (*Intg* | *Bool*)

Objective Evaluate an AS_1 simple expression

Parameters

expr The AS_1 simple expression

sort The spelling of the sort of the expression

Result The Meta-IV types *Intg* if the SDL sort is "Integer" and the Meta-IV domain *Bool* if the SDL sort is "Boolean".

Algorithm For reason of economy in space, this function is not included in the formal definition. Refer to §5.6 of Z.100 for a formal specification of the predefined sorts and to Annex F.3 for a formal definition of the SDL data model.

select-consistent-subset(*systemdefinition*, *genericparameters*) $\triangleq$ (3.14.5)

1 /* *By means of* AS_1 *and the actual subset parameters, select a consistent subset* */

type : *System-definition*$_1$ *External-Information* $\rightarrow$ *Block-identifier*$_1$-**set**

This informal function returns the set of block identifiers denoting the consistent subset. It is used in Annex F.3 for selecting the blocks which should be interpreted. In Annex F.3 it is also checked, that the consistent subset conforms to the properties defined in §3.2.1 of Z.100.

create-unique-name() $\triangleq$ (3.14.6)

1 /* *Create a new* **mk**-*Name*$_0$(*Token*,**nil**) */

type : () $\rightarrow$ *Name*$_0$

This informal function returns an AS_0 *Name*$_0$ which has not previously been used. Note that this function by nature is imperative. A global variable keeping track of the used names is assumed used by the function

check-name-syntax(*string*) $\triangleq$ (3.14.7)

1 /* *Convert a character string to uppercase letters* */

type : *Char** $\rightarrow$ *Char**

First, any sequence of underline character followed by spaces is removed, then any sequence of spaces are replaced by an underline character then the spelling is converted to uppercase and finally it is checked that the resulting spelling conforms to the lexical rules for names as defined in §2.2.1 of Z.100.

4 Deviations From Z.100

After the final approval of the Z.100 recommendation text by SG X (Marts 1988) some few inconsistencies or omissions have been found in the recommendation text. For reasons of completeness of the Formal Definition some decisions in these situations have been taken:

1. §2.7.4 in Z.100 the sentence:
   "However, order is preserved if the two signals are conveyed by identical path connecting the *Originating-process* with the *Destination-process*."
   is interpreted as
   "However, order is preserved if the two signals are conveyed by identical channels connecting the *Originating-process* with the *Destination-process* or if they are defined in the same block."

2. §2.7.5 in Z.100 the sentence:
"The *Decision-answers* must be mutually exclusive."
is interpreted as
"The *Decision-answers* must be mutually exclusive and they must be of the same sort. If the *Decision-question* contains an *Expression* then it must be of the same sort as the *Decision-answers*."

3. §2.7 in Z.100 a new sentence before *Model* is added in order to avoid a circular definition:
"The context of the decision (i.e. the sort) is determined without regard to <answer>s which are <character string>.

4. §3.3 in Z.100 the sentence:
"If the consistent partitioning subset contains a block definition communicating with sub-signals, then the blocks with which it communicates must use the same sub-signals"
is interpreted as
"When selecting the consistent subset, the set of signals on signalroutes connected to an endpoint of a channel must not contain parent signals of contained sub-signals and unless the other endpoint is the system ENVIRONMENT, the set of signals for the first endpoint must be equal to the set of signals on signalroutes connected to the other endpoint."

5. §4.3.3 in Z.100 the sentence:
"The <boolean simple expression> must not be dependent on any definition within the <select definition>.
is modified as follows (in order to guarantee a unique selection and a unique binding of synonyms):

   A Synonym used directly or indirectly (via other synonyms) in <boolean simple expression> of a <select definition> must satisfy the following conditions:

   - It must conform to the visibility rules stated in §2.2.2 even if the synonym definition is not selected.
   - The sort in the <synonym definition> must be qualified with the system level if there exist a visible redefinition of that (predefined) sort, even if the redefinition is local and not selected.
   - There must exist an order of selection which allows all <simple expression>s to be evaluated.

6. §4.10.2 in Z.100 the sentence:
" A signal is a high priority signal if and only if it is mentioned in a <priority input> of a <service definition>."
is interpreted as
" A signal is a high priority signal in a process if and only if it is mentioned in a <priority input> of a <service definition> in the process."

7. §4.10.2 in Z.100 the sentence:
" A <procedure definition> must not have <state>s when the enclosing <process definition> contains a <service definition>."
is extended as
" A <procedure definition> must not have <state>s when the enclosing <process definition> contains a <service definition>. <procedure definition>s visible to more than one service must not contain a VIA construct"

8. §5.6.7.1 in Z.100 the equations:

```
r > 1 == True => a * r > a == True;
r > 1 == True => a / r < a == True;
```

   should be

```
r > 1 == True => a * r >= a == True;
r > 1 == True => a / r <= a == True;
```

to cover the case where a equals zero.

Domain Index

Function Index

Quotation Index

Error Messages

§2.10.2: VIA illegal outside services when service signal routes defined 210
§2.2.2: Character string is not of appropriate sort 142
§2.2.2: Conditional expression does not match the context 154
§2.2.2: Ending name in block definition is different from defining name 54
§2.2.2: Ending name in block sub-structure definition is different from defining name 54
§2.2.2: Ending name in channel definition is different from defining name 57
§2.2.2: Ending name in channel sub-structure definition is different from defining name 58
§2.2.2: Ending name in procedure definition is different from defining name 67
§2.2.2: Ending name in process definition is different from defining name 61
§2.2.2: Ending name in service definition is different from defining name 233
§2.2.2: Ending name in system definition is different from defining name 33
§2.2.2: Identifier is not visible 158
§2.2.2: Local synonym or variable is not of right sort 139
§2.2.2: Multiple appearence of operator definition 111
§2.2.2: Names of formal parameters must be distinct from variable names 61, 67
§2.2.2: No appropriate sort exist for decision action 218
§2.2.2: No import variable matches the context 150
§2.2.2: No sort can be used for value identifier 137
§2.2.2: No synonym, variable or literal matches the sort of the context 139
§2.2.2: One and only one revealed variable must match the view expression 151
§2.2.2: One and only one variable must match the context 160
§2.2.2: Ordering operators are multiple defined 108
§2.2.2: Result sort of operator is illegal for the context 147
§2.2.2: Several variables and synonyms are possible for the context 139
§2.2.2: Signal identifier denotes more than one (sub) signal 159
§2.2.2: Synonym or generator or signal list is recursive defined 158
§2.2.2: The operator term cannot be used in this context 143, 144
§2.2.2: Two definitions in the same scopeunit and same entity class define the same name 51, 66
§2.2.2: Two definitions in the same scopeunit use the same name 64, 65, 68, 69, 117, 119, 121
§2.2.3: Initial number is greater than maximum number of instances 61
§2.2.3: Initial number of instances is less than zero 61
§2.2.3: Maximum number of instances equals zero 61
§2.4.1: Defining names may only be qualified in remote definitions 54, 58, 61, 67, 233
§2.4.1: Ending and starting identifier in the definition are different 37, 38
§2.4.1: No remote definition matches reference 37, 38
§2.4.1: Remote definition is not referenced in the system definition 35
§2.4.1: Remote definitions are not unique 35
§2.4.2: System definition must contain at least one block 33
§2.4.3: Block must contain either one or more processes or a sub-structure definition 54
§2.4.4: Remote number of instances specification does not match process reference 41
§2.4.5: Variable in procedure (or service) cannot be EXPORTED or REVEALED 117
§2.5.2: The endpoints of the signal route are not different 122
§2.5.2: The endpoints of the signal route are not locally defined 122
§2.5.2: The second path must denote reverse direction of the first path 122
§2.5.2: Valid input signal must contain all local signals and no timers 63
§2.5.2: Valid input signal set must be specified when no signal routes are specified 249
§2.5.3: Channel to signal route connection appears twice 125
§2.5.3: Channel to signal route connection is not well-formed 126
§2.5.3: No connection for channel connected to block 125
§2.5.3: Signal route does not appear in exactly one connect 125
§2.5.3: Signal route mentioned in more than one connection 125
§2.5.5: Signal identifiers in signal list are not distinct 124
§2.5: Endpoint of channel is defined in another scopeunit than the channel 57
§2.5: Endpoints of channel must be different 57
§2.5: Second path in channel definition must denote reverse direction of first path 57
§2.6.1.2: No unique corresponding revealed variable of that sort 120

§2.6.3: Ending name of state must be the same as starting name 172
§2.6.3: Ending state name is not allowed in asterisk state 172
§2.6.3: Signals in state are not disjoint 177, 178
§2.6.3: Signals in state node are not well-formed 175
§2.6.3: State names in state must be distinct 173
§2.6.4: Wrong number of parameters in input node 192
§2.6.6: Labels are not distinct 170
§2.6.7.1: Terminator missing in transition 166, 168
§2.6.7.2.1: Name in nextstate must denote a defined state 195, 196, 197, 238
§2.6.7.2.2: Label in join is not defined 195
§2.7.2: Actual parameter length must be equal to formal parameter list length 212
§2.7.2: Created process must be defined in the same block 212
§2.7.3: Actual IN/OUT parameter must be a variable identifier 216
§2.7.3: Length of procedure parameter list must equals the length of formal parameter list 214
§2.7.3: Same sort must be specified for formal and actual IN/OUT parameter 216
§2.7.4: Channel in VIA set is not connected to block 211
§2.7.4: Identifier in output action must denote a signal 208
§2.7.4: If signal routes are specified then channels cannot be mentioned in VIA 211
§2.7.4: Illegal VIA set 210
§2.7.4: Not the right number of arguments to output action 208
§2.7.4: Signal cannot be conveyed by channel in VIA set 211
§2.7.4: Signal in output has no possible receiver 210
§2.7.4: Signal route or channel specified twice in VIA set 210
§2.8: Identifier in reset action is not a timer 203
§2.8: Identifier in set action is not a timer 205
§2.8: Identifier in timer active expression does not denote a timer 152
§2.8: Illegal timer active expression 152
§2.8: Parameter list length in reset action must equal the length specified in the definition 203
§2.8: Parameter list length in set action must equals the length specified in the definition 205

§3.2.2: Block sub-structure does not contain a block definition 54
§3.2.2: Channel connection appears twice 128
§3.2.2: Channel connection is not well-formed 129
§3.2.2: No connection for channel connected to block substructure 128
§3.2.2: Sub-channel does not appear in exactly one connect 128
§3.2.2: Sub-channel mentioned in more than one connection 128
§3.2.3: The block identifiers in the connects do not denote channel endpoints 60
§3.2.3: There must be exactly two channel endpoint connects 60
§3.3: Process uses signals on different refinement levels of the same signal 61

§4.10.1: Service signal route identifier occurs twice in service signal route connection 133
§4.10.1: Signals in service signal routes must be the same as for the connected signal route 133
§4.10.1: The endpoints in the directions of a bidirectional signal route must match 241
§4.10.1: The endpoints of service signal route must be different 241
§4.10.2: All service signal routes must be mentioned in a connect 132
§4.10.2: Illegal service signal route connection 132
§4.10.2: Missing signal route connection in process 132
§4.10.2: More than one service contains an initial transition string 238
§4.10.2: No service signal routes are connected to signal route in VIA 210
§4.10.2: Procedure in decomposition or service has states or import expressions 67
§4.10.2: Process contains both service definitions and timer definitions 116
§4.10.2: Service signals not well-formed 161
§4.10: A decomposition must contain at least one service definition 235
§4.10: Illegal use of high priority signal 208
§4.11: Only one continuous signal may be specified if the priority is omitted 182
§4.11: Several continuous signals with the same priority 185
§4.2.3: The two terms in equation must be of the same sort 103

§4.3.2: Simple expression is not of a predefined sort or contains undefined identifiers 44
§4.3.3: The definitions in select are not allowed in that scopeunit 39
§4.3.3: The selected definition is not allowed in that scopeunit 48
§4.3.4: More than one option answer matches the question 221
§4.3.4: No option answer matches the option question 221
§4.4: Illegal asterisk state 173
§4.7: State has more than one asterisk input or save 177, 178
§4.9: Dash nextstate in initial transition 195

§5.2.1: Ending name in partial type definition is different from defining name 71
§5.2.1: Sort in data type definition have no values 226
§5.2.1: There exist no operators which returns a value of that sort 71
§5.2.2: Literal defined twice in a partial type definition 94
§5.2.2: Operator or literal both defined explicit and by inheritance 73
§5.2.3: Inconsistent use of implicitly quantified value name 141
§5.2.3: Value identifier must not be qualified 141
§5.2: Length of parameter list is not correct for an operator 147
§5.4.1.10: Two structure fields have the same name 113
§5.4.1.11: Literal defined twice in renaming 76
§5.4.1.11: Literal in literal renaming is not defined in the parent sort 76
§5.4.1.11: Literal renamed twice 76
§5.4.1.11: Operator in operator renaming is not defined in the parent sort 78
§5.4.1.11: Operator renamed twice 78
§5.4.1.11: Operator with an exclamation is mentioned in an operator renaming 78
§5.4.1.11: Sort is based on itself 73
§5.4.1.12.2: CONSTANT actual generator parameter must be a term 89
§5.4.1.12.2: LITERAL actual generator parameter must be a literal signature 89
§5.4.1.12.2: Lengths of actual and formal parameter lists in generator must be the same 84
§5.4.1.12.2: OPERATOR actual generator parameter must be an operator signature 89
§5.4.1.12.2: TYPE actual generator parameter must be a identifier 89
§5.4.1.12: Ending name in generator definition is different from defining name 70
§5.4.1.12: Generator constant parameter used in literal signature 87
§5.4.1.12: Generator formal name is qualified 87
§5.4.1.12: Name class cannot be used in equations 87
§5.4.1.13: Sort of synonym cannot be uniquely determined 93
§5.4.1.14: Length of character strings in regular interval is not equal to one 98
§5.4.1.14: Repetition name in name class must denote a natural number 98, 99, 100
§5.4.1.15: Illegal use of the SPELLING operator 148
§5.4.1.7: Error term is part of restriction 105
§5.4.1.7: Error term is used in a composite term 135
§5.4.1.9.1: Ordering operator is not defined for the sort of the range condition 92
§5.4.1.9: Ending name in syntype definition is different from defining name 90
§5.4.1.9: Syntype is based on itself 90
§5.4.1.9: Syntype is defined in terms of itself 256, 258
§5.5.3.3: More than one default assignment 84
§5.5.3: None or multiple expansions of modify assignment 201
§5.5.4.1: NOW expression must be used where TIME values are allowed 151
§5.5.4.3: PID expression is used in wrong context 153
§5.5.4: Imperative operator cannot be used in constant expression 149

ISBN 92-61-03781-X